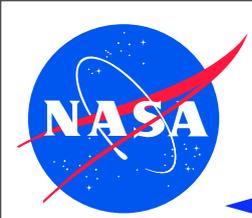


**JPF v7**

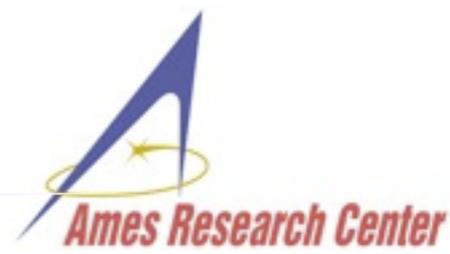
~

**just what we needed**

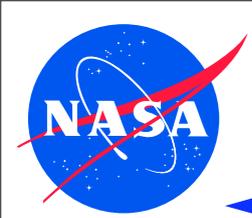
Peter C. Mehlitz  
SGT / NASA Ames Research Center  
<[peter.c.mehlitz@nasa.gov](mailto:peter.c.mehlitz@nasa.gov)>



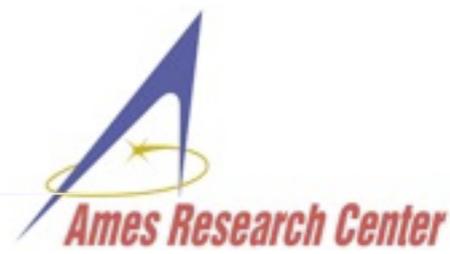
# Talk Motivation



- ◆ this is *not* the general v7 release yet, you can still help or at least cast your vote
- ◆ **v7 will require porting extensions**
- ◆ talk mostly intended to
  - prepare you for the related API and functional changes
  - enable you to start porting your extensions and test v7
  - get feedback relevant for v7 *before* everybody is worn out from the workshop/conference



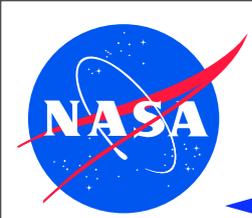
# Roadmap



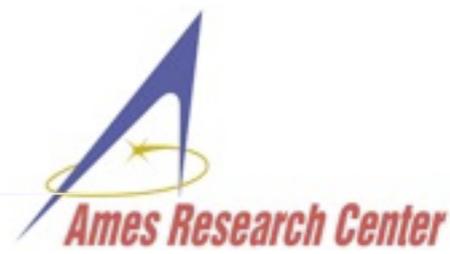
- ◆ presentation
  - more of a code review than elaborate presentation (lots of code snippets)

~ Break ~

- ◆ free form
  - discussion of presented v7 features
  - discussion of feature requests
  - optional hands-on: installing v7, porting of extension



# How to get v7



- ◆ you already have it, you are just in the 'default' branch when you clone the repo:

```
> hg clone http://babelfish.arc.nasa.gov/hg/jpf/jpf-core
```

```
..
```

```
> hg branches
```

```
v7-dev          956:c40104e37b12
```

```
default        791:ff64c0a66f36
```

```
> hg branch
```

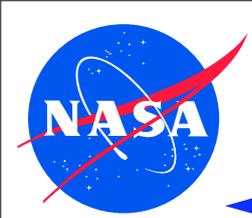
```
default
```

```
> hg update v7-dev
```

```
693 files updated, 0 files merged, 272 files removed, 0 files unresolved
```

```
> hg branch
```

```
v7-dev
```



# Statistics



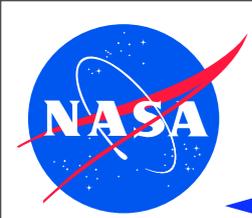
- ◆ massive change, between 06-22-12 and now:

```
> hg diff --stat -r 724 -r 956
```

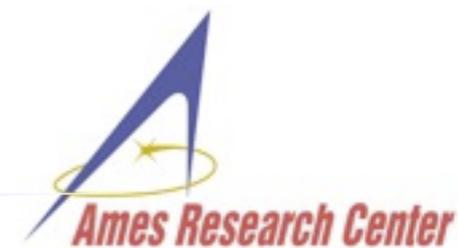
```
....
```

```
968 files changed, 72081 insertions(+), 63819 deletions(-)
```

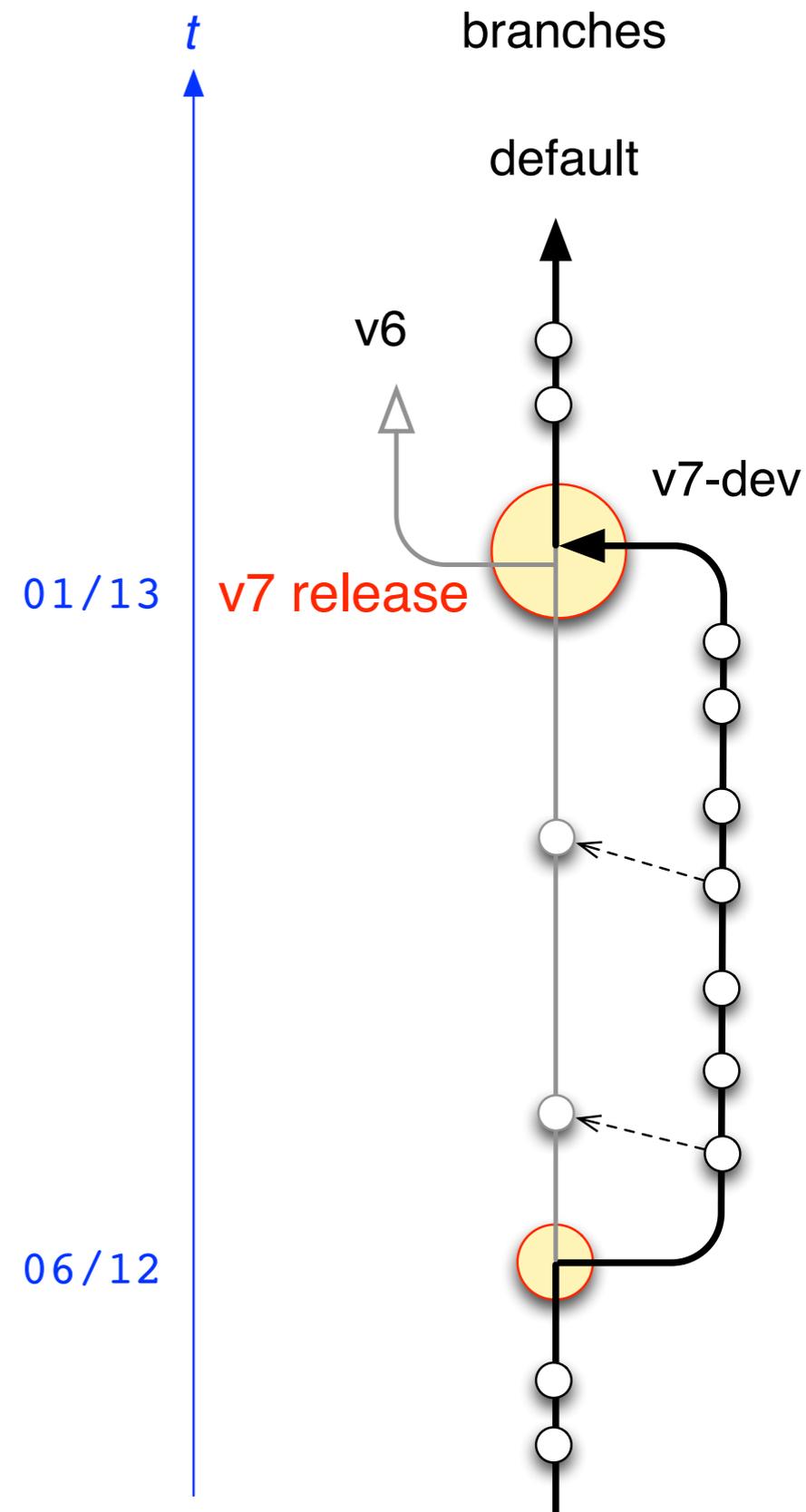
- ◆ total as of r956: 936 files, 114411 loc
- ◆ this really is going to be a major release

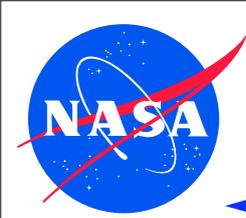


# Release Plan

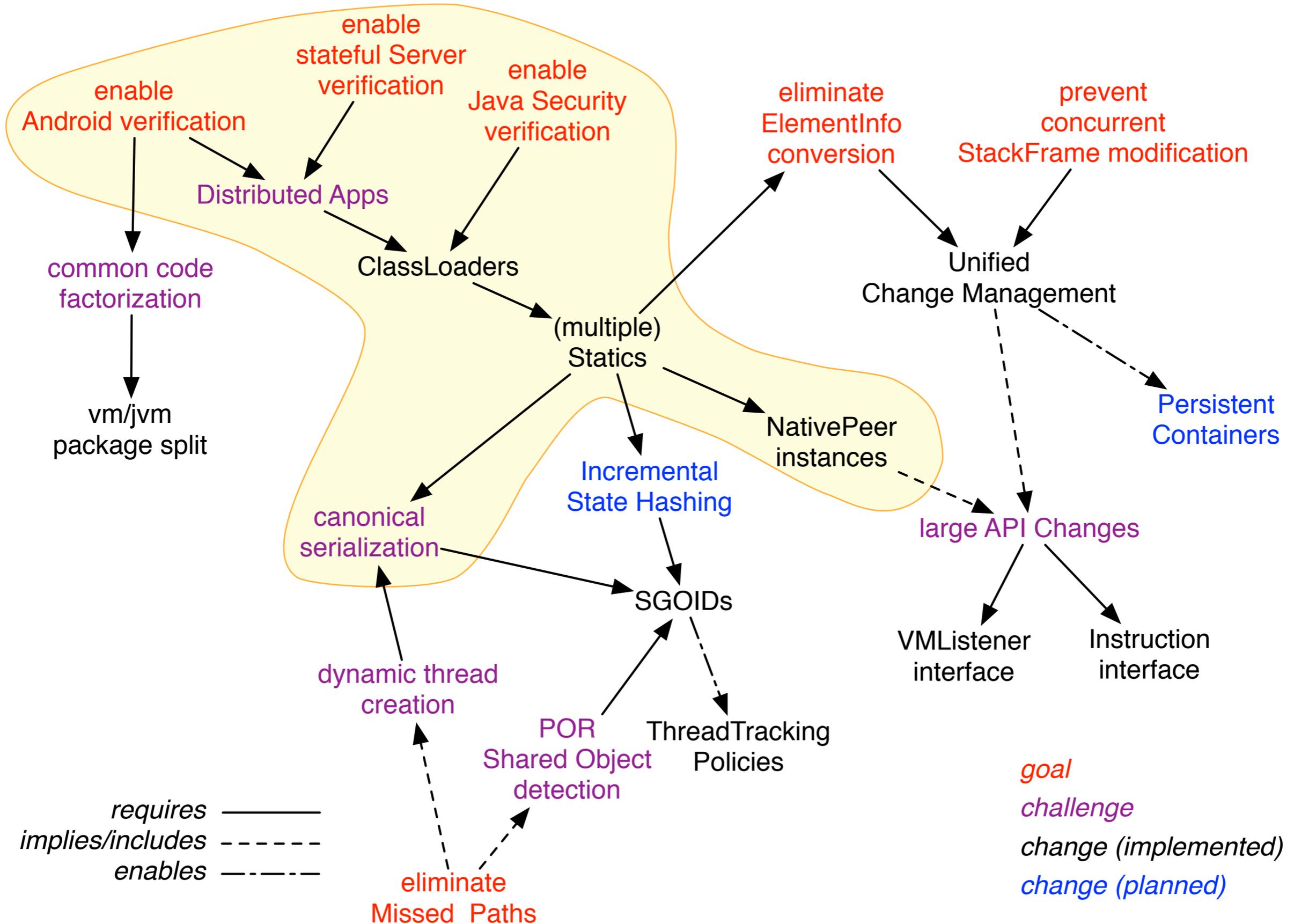


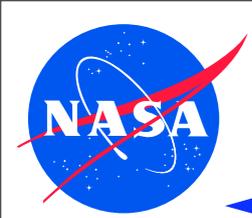
- ◆ No Panic! Unless you do a `hg update v7-dev` you are still in default (v6)
- ◆ active development branch is v7-dev
- ◆ currently only critical bugs are back-ported to default (v6)
- ◆ nothing gets lost, we will create a v6 branch before turning v7-dev into default
- ◆ expected sometime early next year
- ◆ BUT: depends on your test input  
→ start using v7-dev



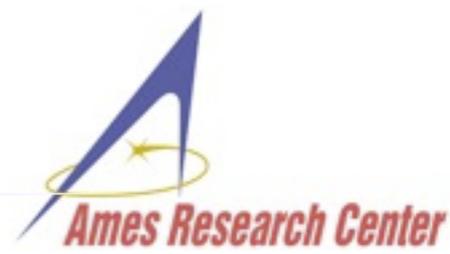


# in medias res - v7 Change Map

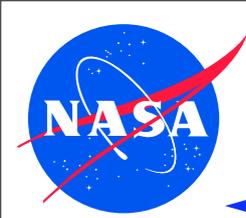




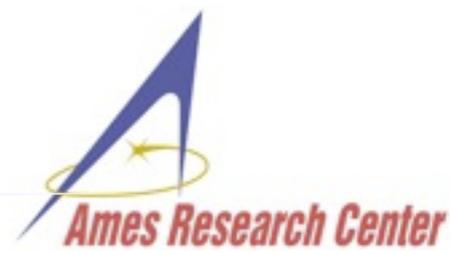
# ClassLoaders - Why?



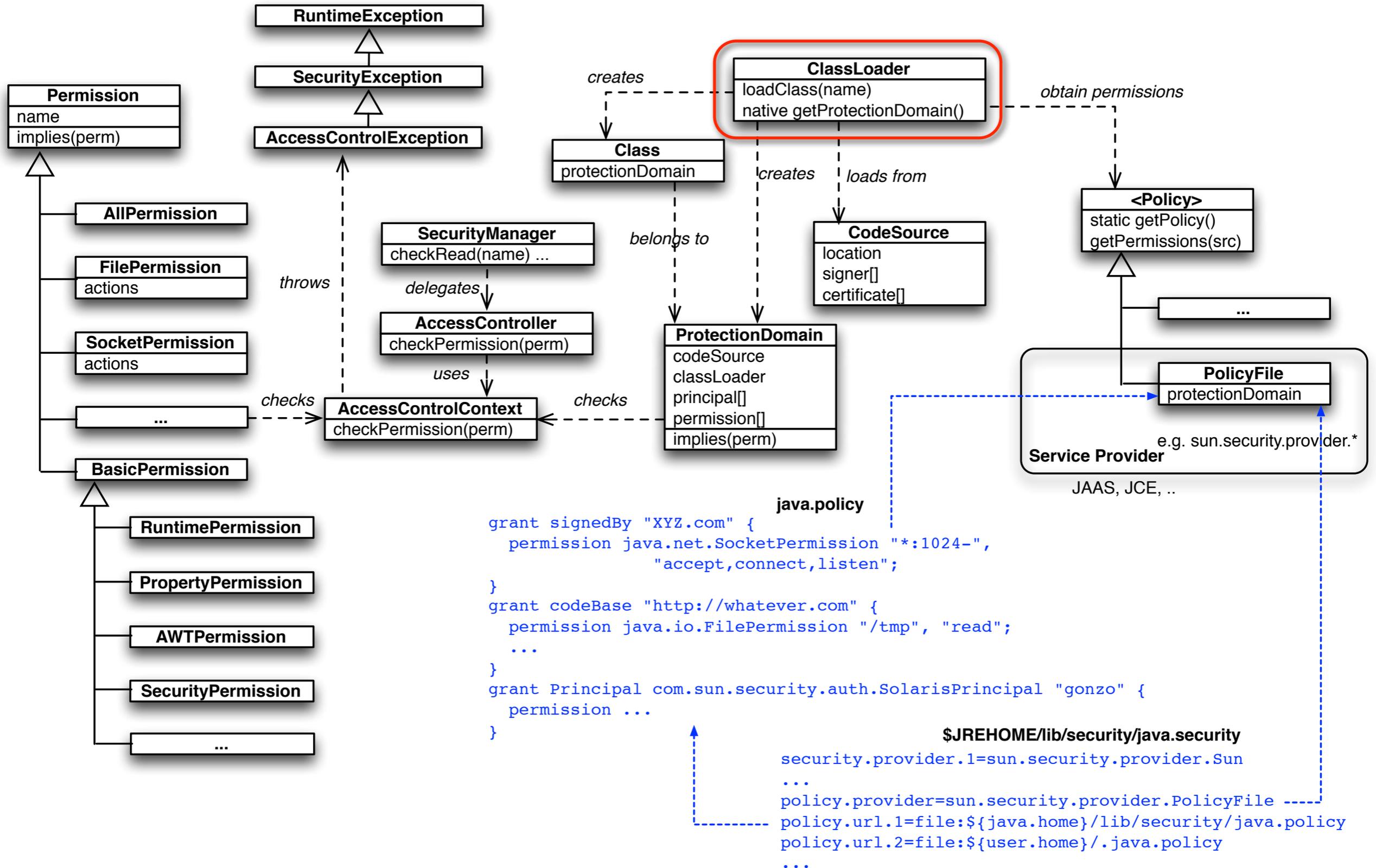
- ◆ support for user defined class loaders last missing major VM function
- ◆ .. but how relevant outside of low level 3rd party library verification?
  
- ◆ production code
  - CL verification (lookup/delegation logic)
  - type confusion (non-parent delegating CLs)
  - on-the-fly instrumentation
  - basis for **Java Security Policies** (CL creates and assigns to ProtectionDomains)
  
- ◆ **distributed application model checking**  
(mapping applications into JPF threads with separate type namespaces)

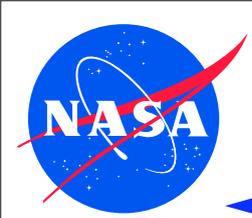


# ClassLoaders - Security

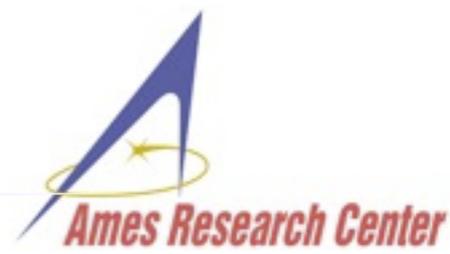


◆ ClassLoader is essential Java Security Infrastructure component

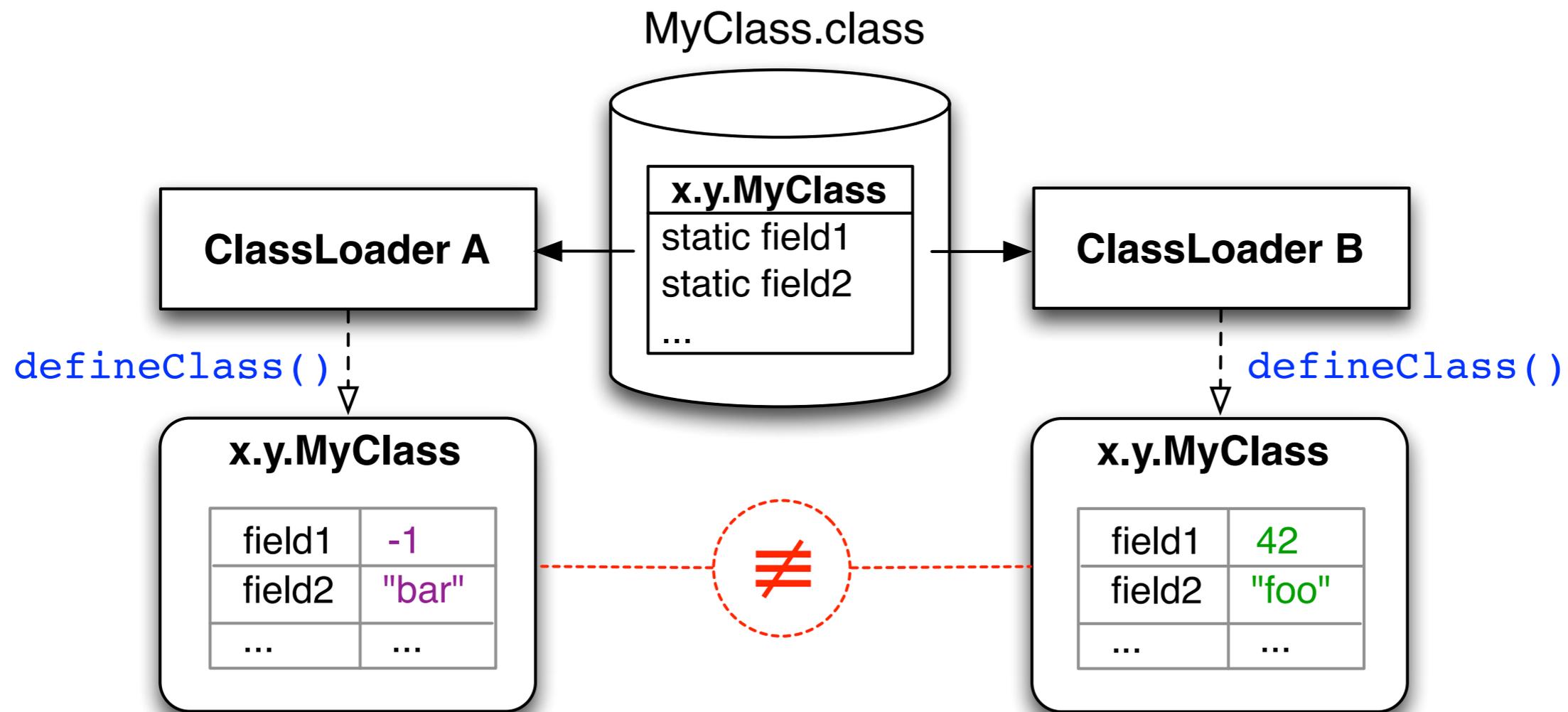




# ClassLoaders - Distributed Apps



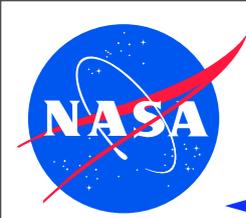
- ◆ Java type = classname + defining ClassLoader instance
- ◆ types have separate static field values



```
MyClass.field1 = -1  
o1 = new MyClass()
```

```
MyClass.field2 = 42  
o2 = new MyClass()
```

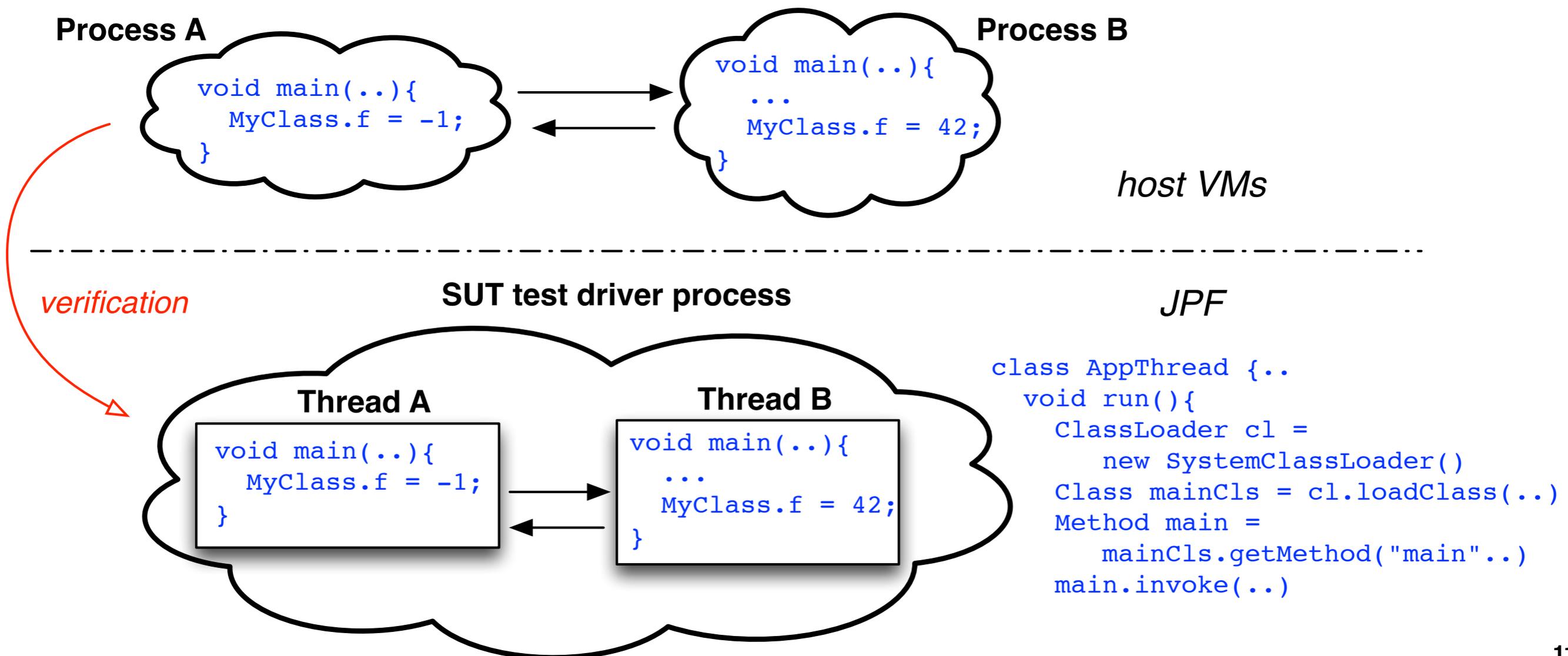
```
o1.getClass() != o2.getClass()
```

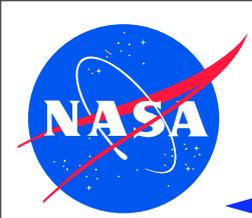


# ClassLoaders - Distributed Apps

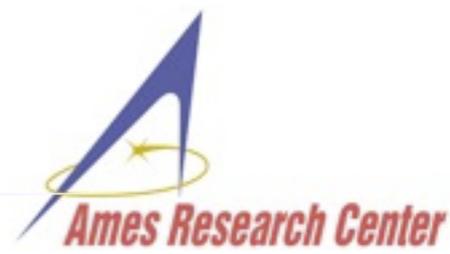


- ◆ different ClassLoader type-namespaces useful to model static fields of “same” classes in different processes
- ◆ run different apps (native processes) as different threads within the same SUT, using separate ClassLoaders for each thread
- ◆ needs a non-standard VM extension: instantiable SystemClassLoader



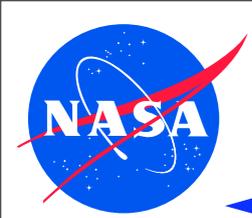


# ClassLoaders - Statics



- ◆ v6: single simulated ClassLoader → all static fields and class states kept in single **StaticArea**
- ◆ v7: any number of ClassLoaders → each one has its own **Statics** object (Area/StaticArea is finally gone)
- ◆ Statics interface closely follow Heap: abstract container for ElementInfos
- ◆ concrete implementation can be configured

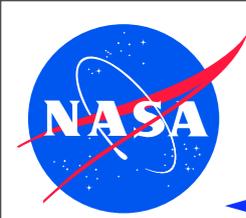
```
class ObjVectorStatics implements Statics {  
    StaticElementInfo newClass(ClassInfo ci, ThreadInfo ti) {  
        int id = computeId(ci);  
        StaticsElementInfo sei = new StaticsElementInfo(ci, id);  
        elements.set(id, sei);  
        return sei;  
    }  
    StaticElementInfo get(int id) { return elements.get(id); }  
    StaticElementInfo getModifiable(int id) {...}  
    Memento<Statics> getMemento() {...}  
    ...  
}
```



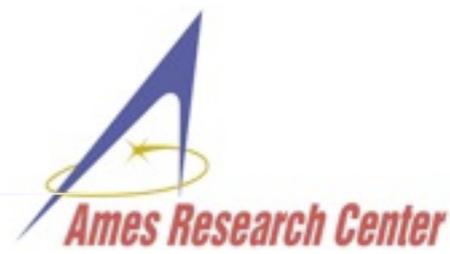
# ClassLoaders - NativePeer Instances



- ◆ required to ensure separation of peer data for different ClassInfo instances for the same class name (ClassLoaders)
- ◆ enables refactoring into peer class hierarchies
- ◆ so far, this is only one peer instance per ClassInfo (i.e. type) instance, NOT per object instance
- ◆ possible to set peer dynamically (e.g. during abstraction refinement)



# ClassLoaders - NativePeer Instances

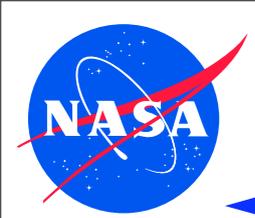


- ◆ v6: native peers used to be statics only, using modifiers (public && static && !final) to identify native methods:

```
class JPF_x_y_MyPeer {  
    ..  
    public static boolean foo__I__Z (MJIEnv env, int objref, int arg){  
        ..  
    }
```

- ◆ v7: now they are NativePeer instances that can have static and instance native methods, identified by @MJI annotation

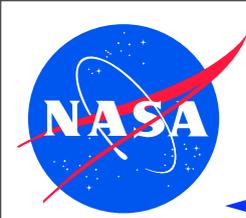
```
class JPF_x_y_MyPeer extends NativePeer {  
    ..  
    @MJI  
    public boolean foo__I__Z (MJIEnv env, int objref, int arg){  
        ..  
    }
```



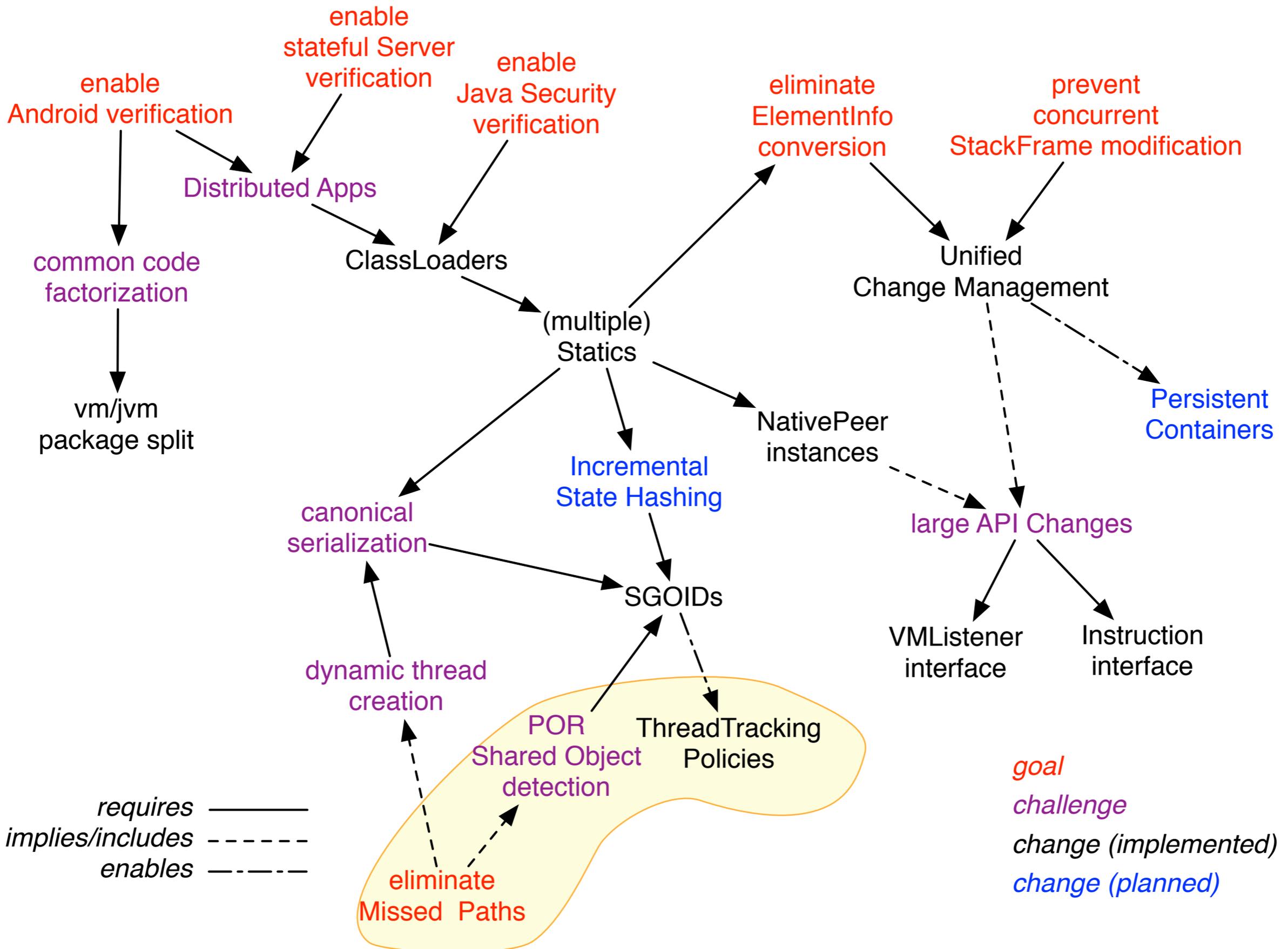
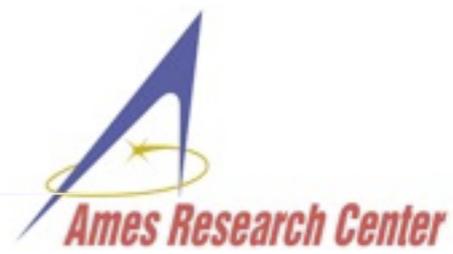
# ClassLoaders - State Matching Challenge

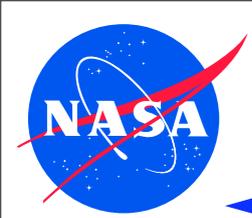


- ◆ different Statics objects holding StaticElementInfos → requires canonical enumeration order for serialization (state matching)
- ◆ each ClassLoader is a SUT (Heap) object
- ◆ search global Heap object reference values can be used to construct the canonical order → **SGOIDS**

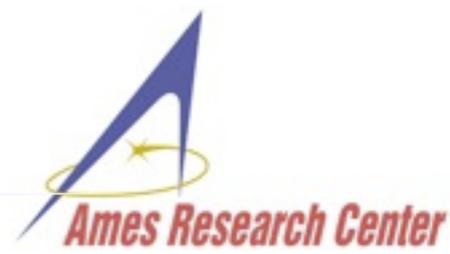


# v7 Change Map





# MissedPaths



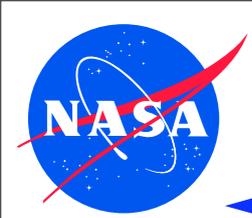
- ◆ surprisingly easy to come by with, but rare in production code
- ◆ depends on lack of interaction between threads

```
public class MissedPath extends Thread {
    static class X { boolean pass; }
    X myX;

    public static void main(String[] args){
        MissedPath mp = new MissedPath();
        mp.start();           // (0) both threads runnable

        X x = new X();       // referencingThreads initialized to {M}
        mp.myX = x;          // (1) myX only shared for T after this was executed
        // .. error only shows if main thread in this section
        x.pass = true;       // (2) error in T only shows before this was executed
    }                        // (3) main terminated

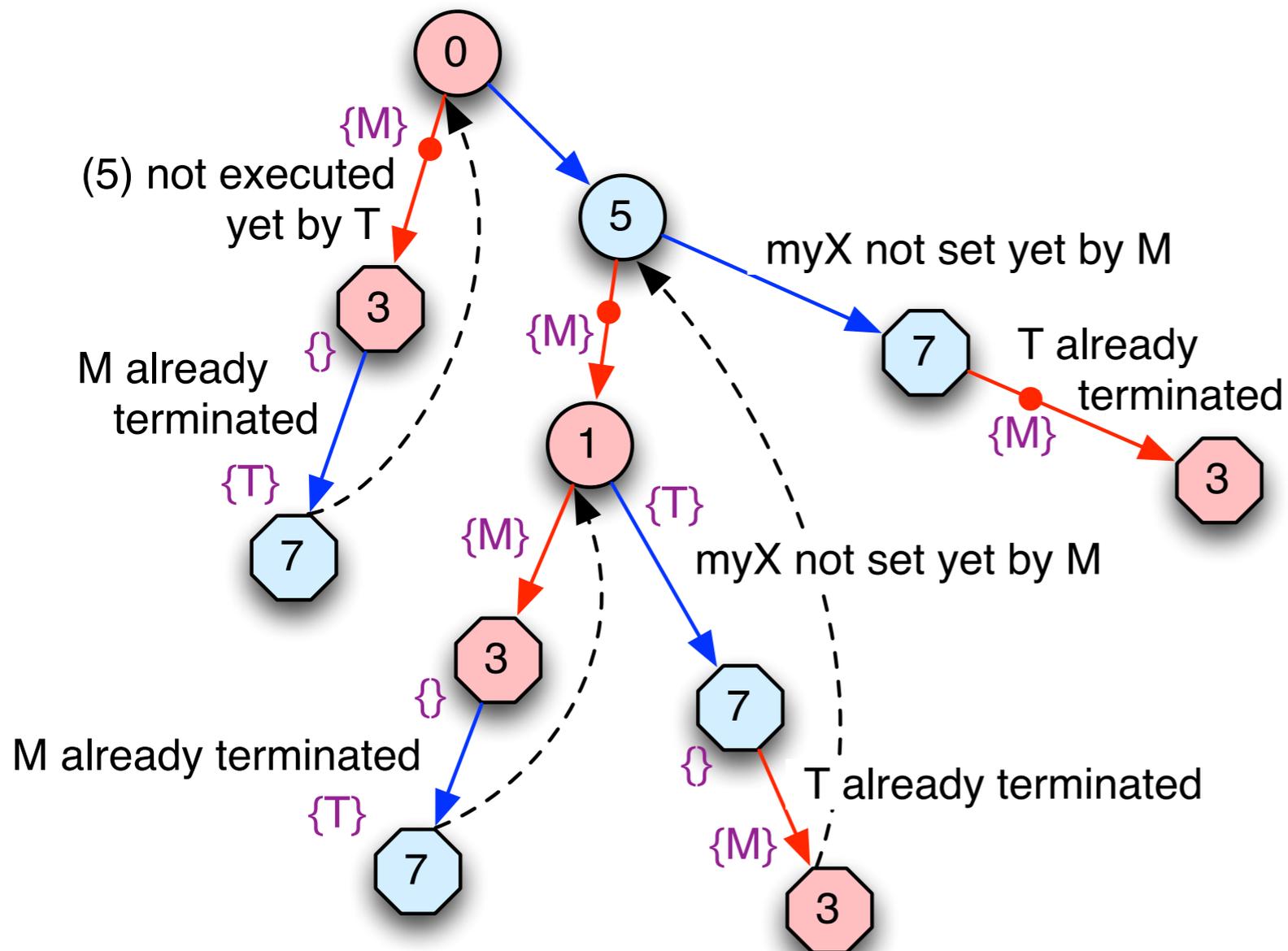
    public void run(){      // (4) thread started
        if (myX != null){   // (5) myX only shared for M after this was executed
            if (!myX.pass){ // (6) won't fail unless main is between (1) and (2)
                throw new RuntimeException("gotcha");
            }
            ...             // (7) thread terminated
        }
    }
}
```



# MissedPath



- ◆ we never get a CG on (2):
  - either T is already terminated, or
  - (5) not executed yet and hence 'x' not shared



```
X myX;
```

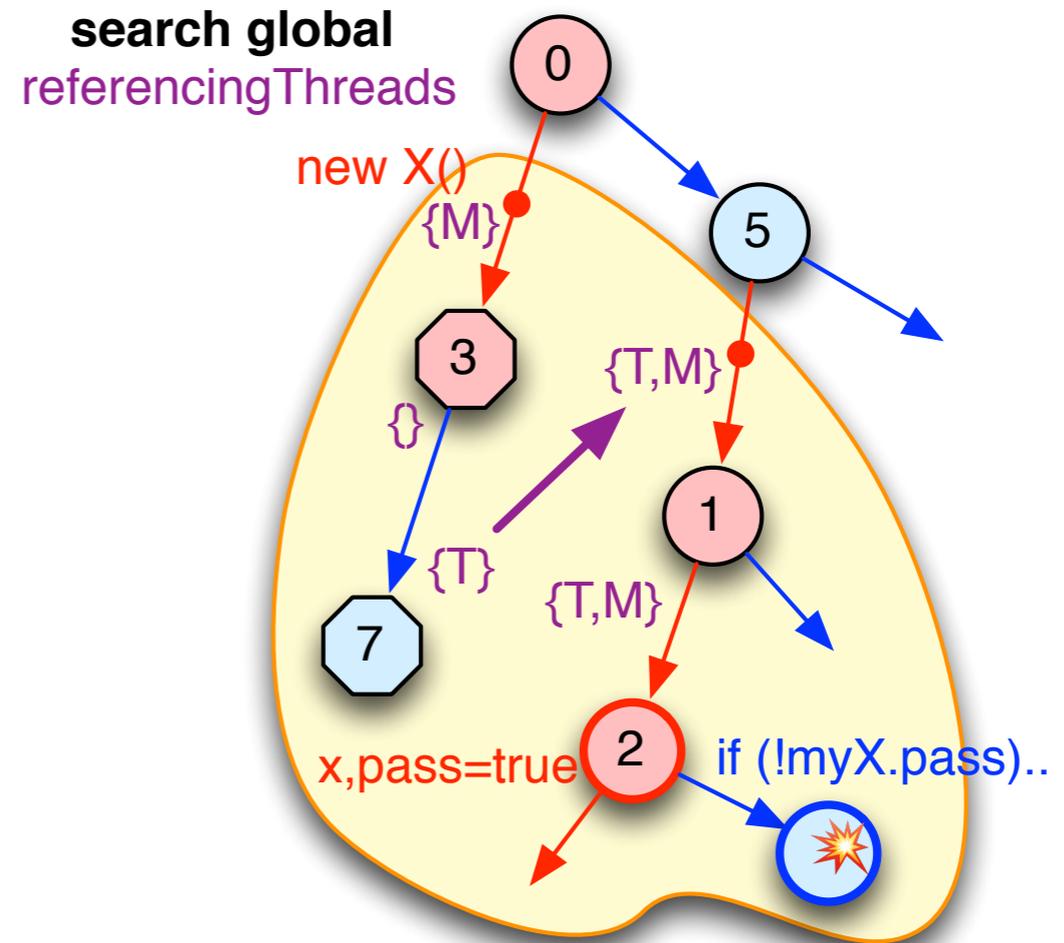
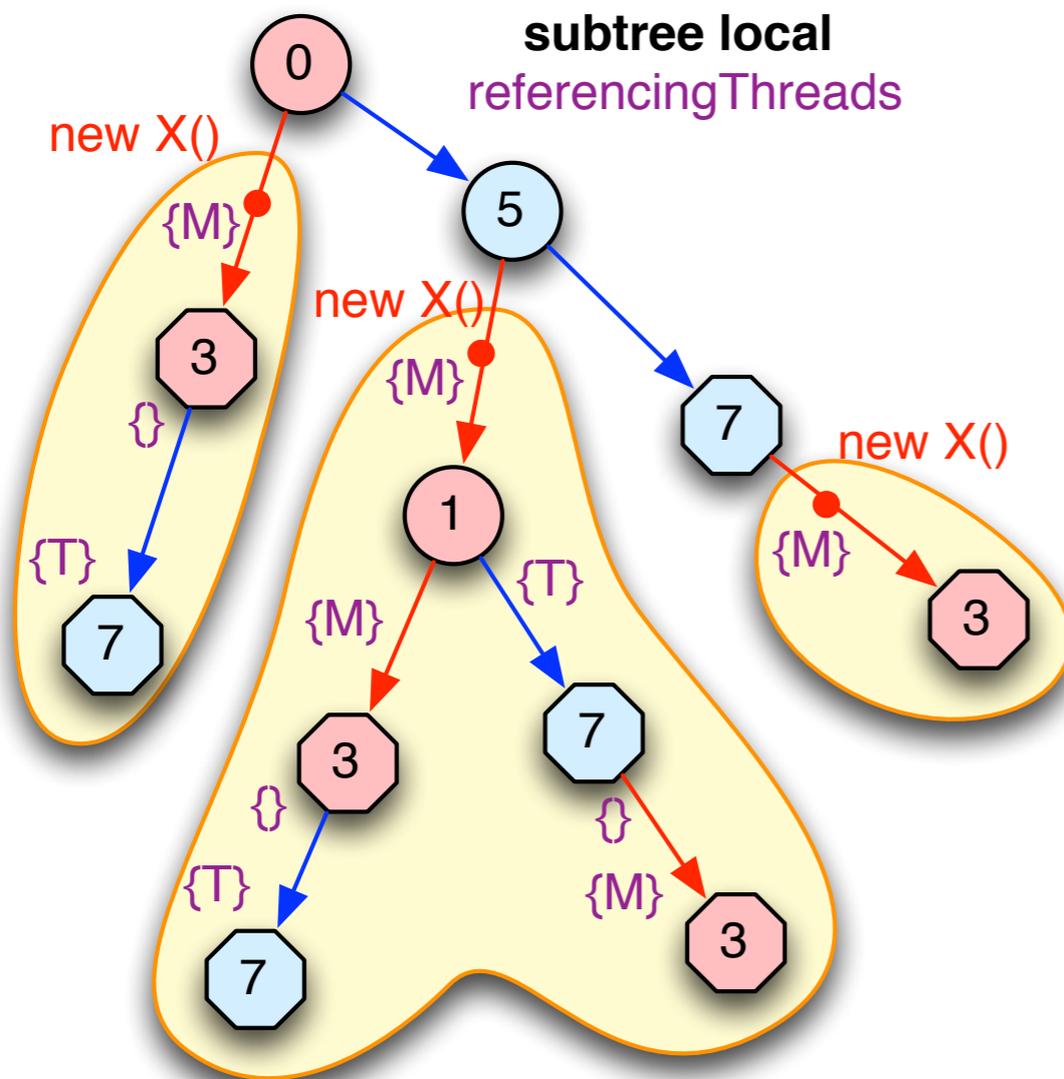
thread M

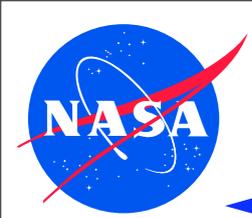
```
main() {
    MissedPath mp =
        new MissedPath();
    mp.start();      // (0)
    X x = new X();
    mp.myX = x;      // (1)
    x.pass = true;   // (2)
}                  // (3)
```

thread T

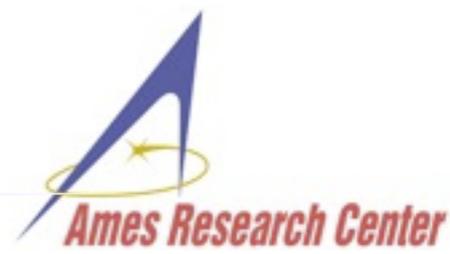
```
run() {
    if (myX != null) // (5)
        if (!myX.pass) // (6)
            throw .. // (*)
}                  // (7)
```

- ◆ problem is *not* removing terminated threads from referencingThreads set (there might be nothing to backtrack to that triggers the defect)
- ◆ problem is *not* missing propagation upon putfield (would again require expensive heap traversal upon assignment)
- ◆ problem is initialization of referencingThreads upon object creation

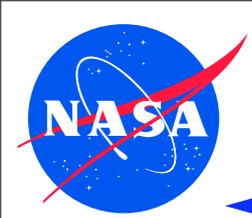




# MissedPaths



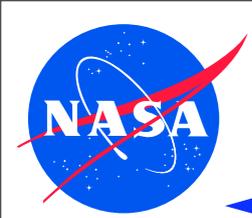
- ◆ challenge: we need to find out if an object was already created on a previous path, to keep referencingThreads search globally
- ◆ solution: **SGOIDS** (Search Global Object Ids)
- ◆ once objects can be identified throughout the whole search, rest is simple: initialize/update referencingThreads set by configured **ThreadTrackingPolicy** object



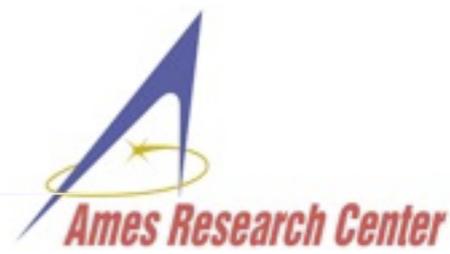
# POR and ThreadTrackingPolicy



- ◆ v7: thread access tracking and POR (sharedness) are separated
- ◆ thread reference tracking:
  - uses abstract type `ThreadInfoSet` (to allow different member storage formats)
  - stored in `ElementInfo.referencingThreads`
  - created, stored/restored through `ThreadTrackingPolicy` singleton
  - used to update `ATTR_SHARED` flag in `ElementInfo.attributes`
- ◆ sharedness check (POR) uses **only** `ATTR_SHARED` flag
- ◆ flag can be set explicitly:  
`Verify.setShared (Object o, boolean isShared)`
- ◆ automatic update of flag from `ThreadInfoSet` can be suspended:  
`Verify.freezeSharedness (Object o, boolean freeze)`



# POR and ThreadTrackingPolicy



- ◆ v6: previous implementation in ElementInfo (e.g. called from InstanceFieldInstruction):

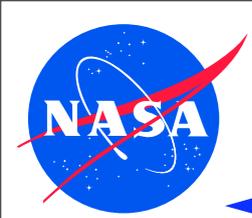
```
class ElementInfo .. {
    IntSet refTid;

    boolean checkUpdatedSharedness(ThreadInfo ti){
        .. refTid.add(ti);
        return refTid.cardinality() > 1;
    }
}
```

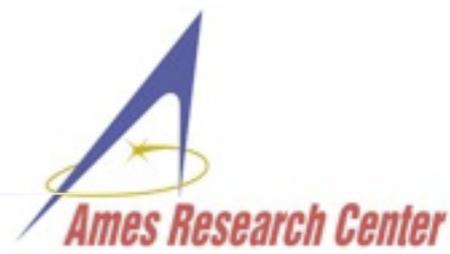
- ◆ v7: BEWARE - new implementation can return new ElementInfo object (to ensure proper copy-on-write for already stored objects)

```
class ElementInfo .. {
    ThreadInfoSet referencingThreads;

    ElementInfo getInstanceWithUpdatedSharedness (ThreadInfo ti) {
        referencingThreads.add(ti);
        if (ThreadTrackingPolicy.getPolicy().isShared(referencingThreads) {
            if ((attributes & ATTR_SHARED) == 0) {
                ElementInfo ei = getModifiableInstance();
                ei.attributes |= (ATTR_SHARED | ATTR_ATTRIBUTE_CHANGED);
                return ei;
            }
        }
    }
}
```



# POR and ThreadTrackingPolicy

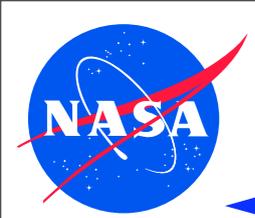


## ◆ v6: old usage pattern

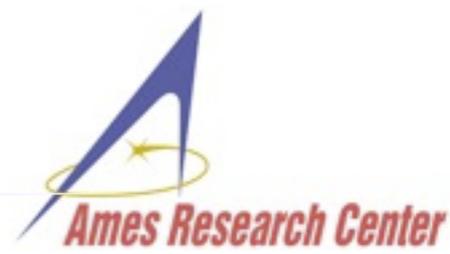
```
class InstanceFieldInstruction .. {
    boolean isSchedulingRelevant (ThreadInfo ti, int objRef) {
        ..
        ElementInfo ei = ti.getElementInfo(objRef);
        if (!ei.checkUpdatedSharedness(ti){
            return true;
        }
        ...
    }
}
```

## ◆ v7: new usage pattern

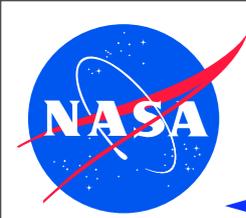
```
class InstanceFieldInstruction .. {
    boolean isSchedulingRelevant (ThreadInfo ti, int objRef) {
        ..
        ElementInfo ei = ti.getElementInfo(objRef);
        ei = ei.getInstanceWithUpdatedSharedness(ti);
        if (!ei.isShared()){
            return false;
        }
        ...
    }
}
```



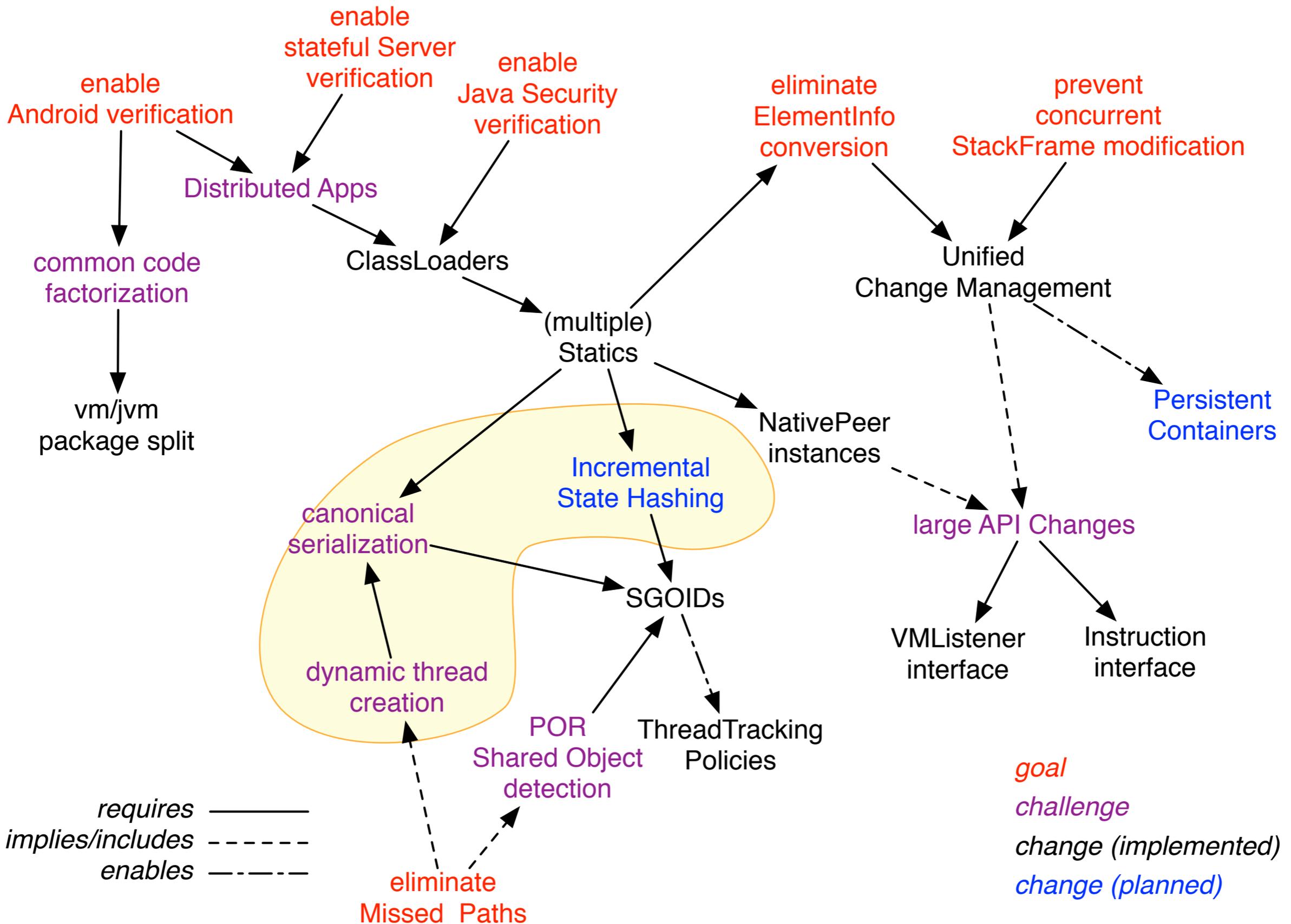
# POR and ThreadTrackingPolicy - v7

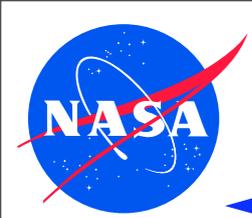


- ◆ OverlappingContenderPolicy
  - old (v6) mode
  - assumes there are enough interactions between contending threads *after* object creation to expose defects
  - referencingThread sets are initialized upon (each) object creation
  - efficient
  - can miss errors
  
- ◆ GlobalTrackingPolicy
  - referencingThread sets are search global (identified by SGOIDs)
  - usually larger state space (e.g. RobotManager: 2762 → 9788)
  - no missed errors due to missing shared field access

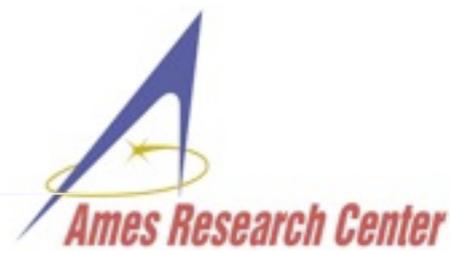


# v7 Change Map

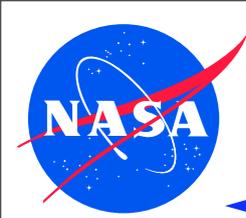




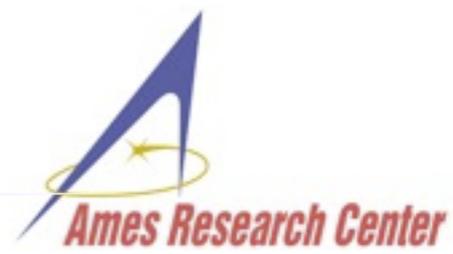
# State Matching Problems - v6



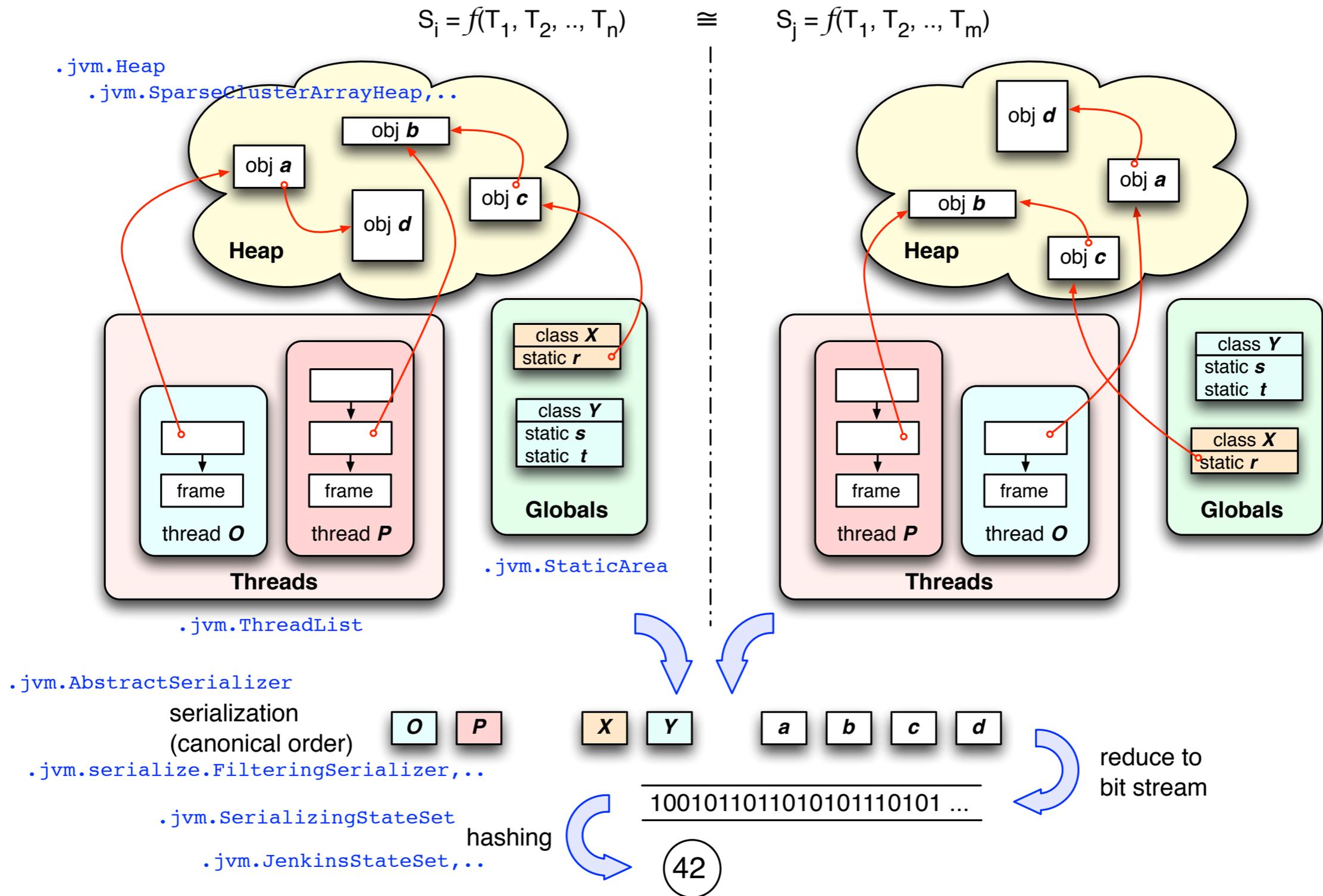
- ◆ performance critical - requires complex logic (caches, ReferenceQueue, alternating ElementInfo.sid, JenkinsStateSet etc.)
- ◆ fragile in terms of Thread enumeration (same would hold for Statics): different serialization order of otherwise identical elements → state leak
- ◆ potentially huge amount of redundant computation (re-hashes all unchanged ElementInfos, StackFrames)

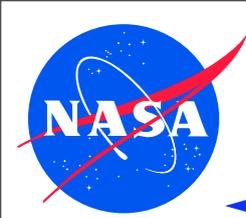


# State Matching - v6



- ◆ v6: state matching with CFSerializer uses (mostly) canonical root set (ThreadList, StaticArea) to serialize/hash whole program state

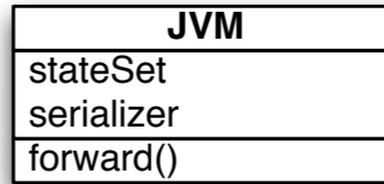




# State Matching - v6



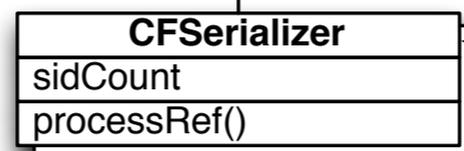
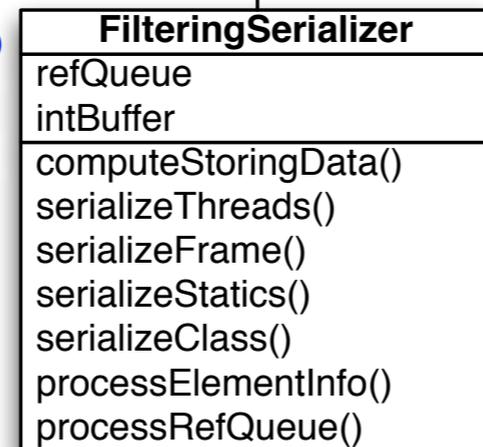
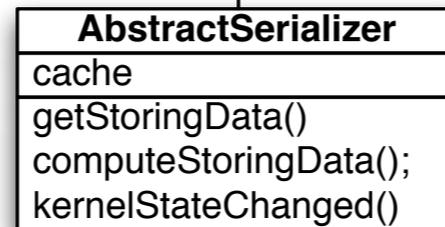
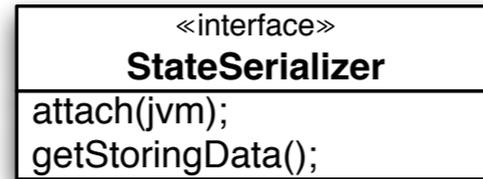
## ◆ complex infrastructure



```

JVM(conf){..
  serializer =
    conf.get("vm.serializer.class")
  stateSet =
    conf.get("vm.storage.class")
}

```



```

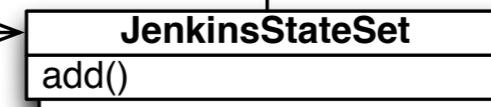
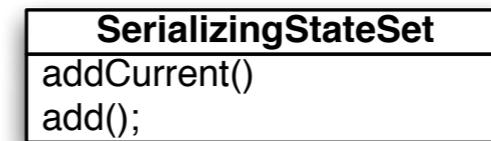
forward(){..
  stateSet.addCurrent(); ..
}

```

```

addCurrent() {..
  add(serializer.getStoringData());
}

```



```

getStoringData() {..
  if (cache==null){
    cache = computeStoringData()
    ks.pushChangeListener(this)
  }
  return cache()
}

```

```

computeStoringData() {..
  intBuffer.clear(); refQueue.clear()
  serializeThreads()
  serializeStatics()
  processRefQueue()
  return intBuffer.toArray()
}

```

```

kernelStateChanged() {
  cache = null
}

```

```

processElementInfo() {..
  fmask = getFilterMask()
  int[] values = getFieldValues()
  for (i<values.length; i++){
    if (!isFiltered(fmask,i)){
      if (isRef(i))
        processRef(values[i])
      else
        intBuffer.add(values[i])
    }
  }
}

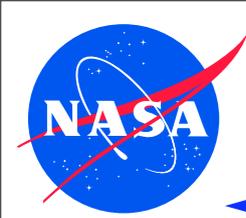
```

*implements Heap Symmetry  
(storing canonical order of reference  
not reference value itself)*

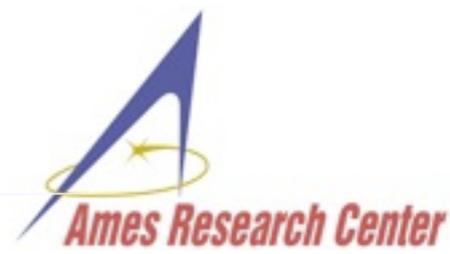
```

processRef(int r) {..
  ElementInfo ei=heap.get(r);
  if (ei.getSid()==0)
    ei.setSid(sidCount++)
  intBuffer.add( ei.getSid())
}

```



# ThreadInfo Enumeration

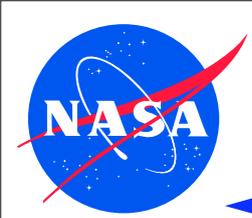


- ◆ similar problem for ThreadInfos: need to be enumerated in canonical order during state matching (with serialization)
- ◆ similar pattern: each thread associated with its own SUT  
java.lang.Thread object
- ◆ java.lang.Thread has public getId(), but
  - id can be re-used (e.g. based on the number of live threads)
  - even if not reused, can depend on order of (non-deterministic) thread creation → state leak

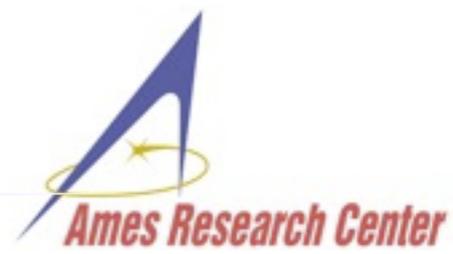
```
class MyApp .. {
    static void main(..){
        new ConsumerThreadPool().start();
        new ProducerThread().start();
        ...
    }
    class ConsumerThreadPool extends Thread {
        void run(){...
            threads[i] = new ConsumerThread(); threads[i].start(); ...
        }
    }
}
```

id = f(order of creation)  
depends on scheduling

- ◆ search global Heap object reference values can be used to construct the canonical order → **SGOIDS**



# Incremental State Hashing



- ◆ better approach would be incremental state hashing:

$$H(S') = H(S) - \sum H(C_i) + \sum H(C_i')$$

$C_i$  : changed state component (ElementInfos, StackFrames)

- ◆ required commutative property can be achieved by giving each byte in each  $C$  a search global, unique “virtual address”, and associate this virtual address with a constant random factor

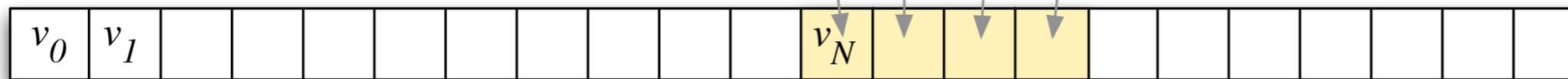
```
class X {
    byte b0, b1, b2, b3;
```

$$N := \text{SGOID}(x1)$$

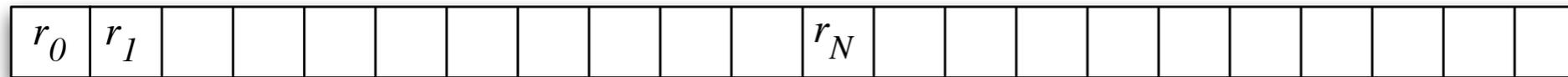
```
foo() {
    ... x1 = new X()
```

$$H(x1) = \sum_{i=N}^{N+3} r_i v_i$$

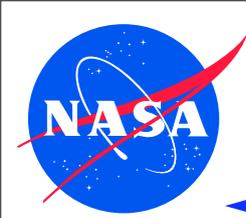
$x1$  always allocated at address  $N$



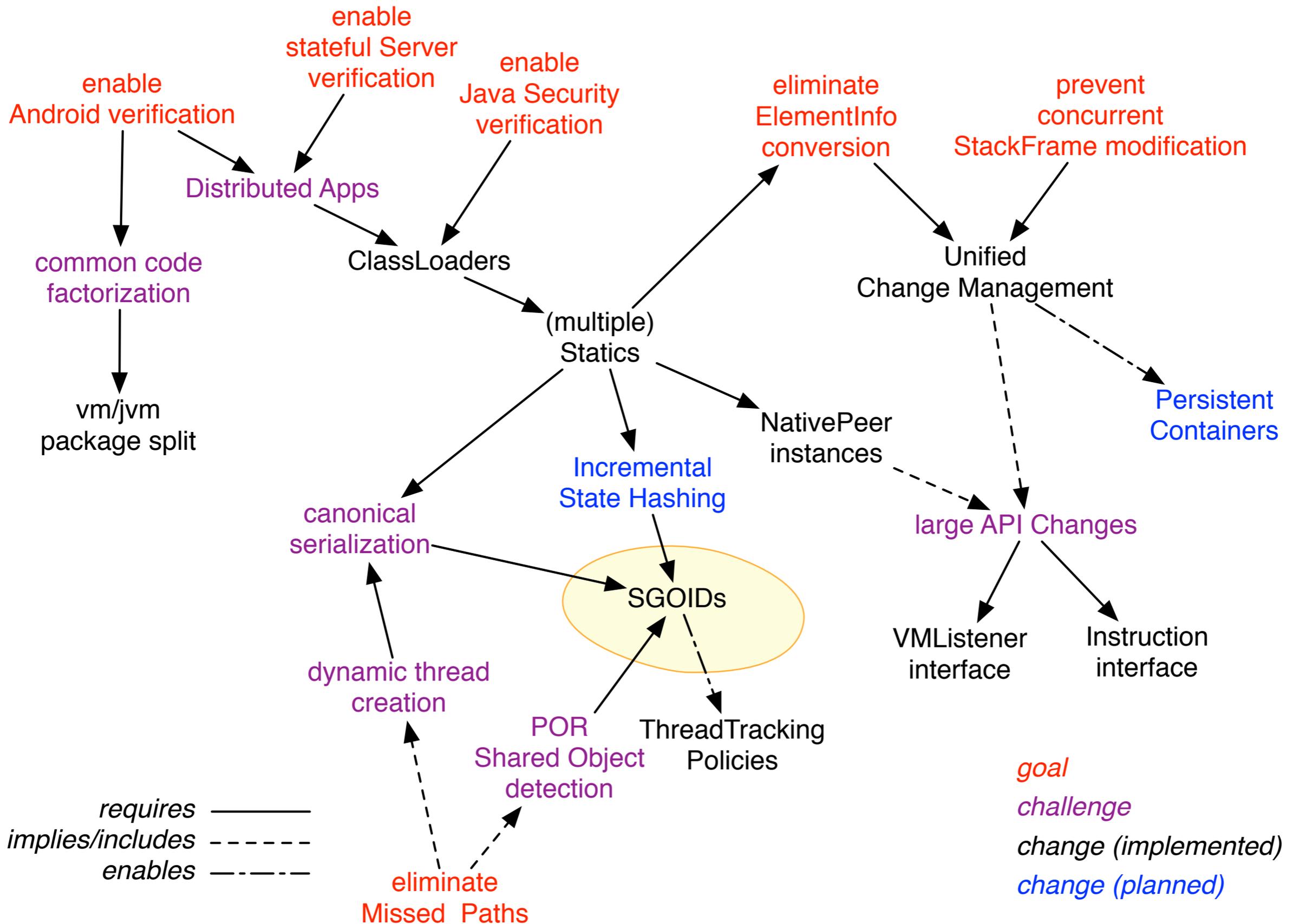
values



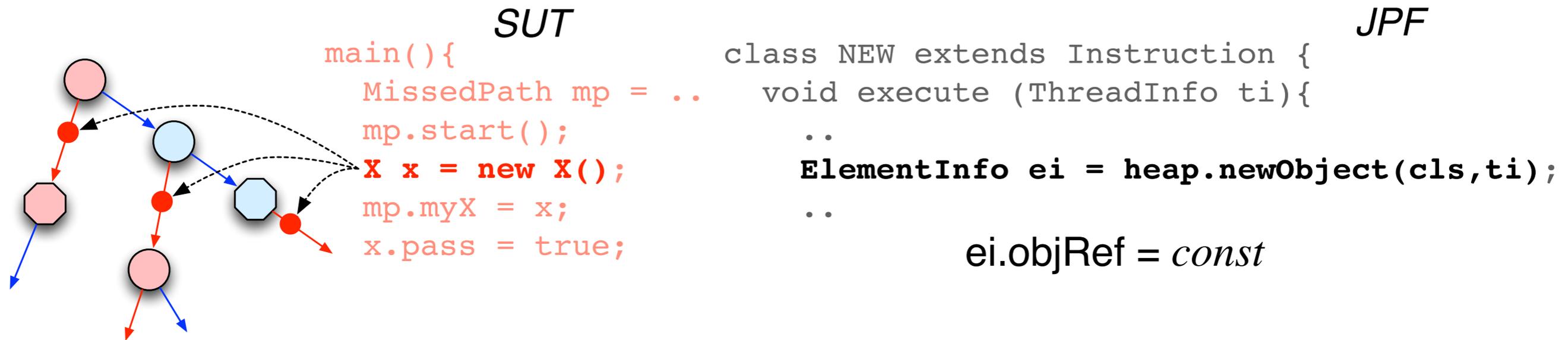
random factors



# v7 Change Map



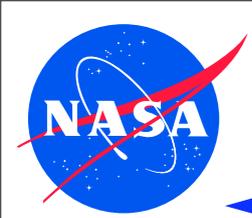
- ◆ ensure same reference value for same object along all allocation paths



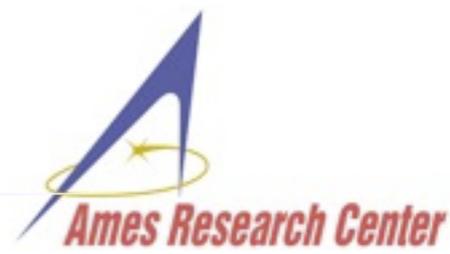
- ◆ but what means “same” object?
- ◆ moreover, different allocation types:
  - SUT: corresponding, explicit NEW instruction
  - System: automatic allocation by VM (main thread, SystemClassLoader, class objects,..)
 

```
.. ci = cls.resolveReferencedClass(cname); ..
```

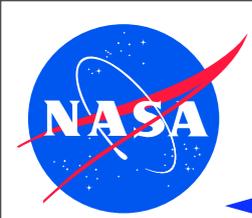
*beware*: can happen non-deterministically (class resolution based on scheduling sequence)
  - SUT/System combinations (newString)



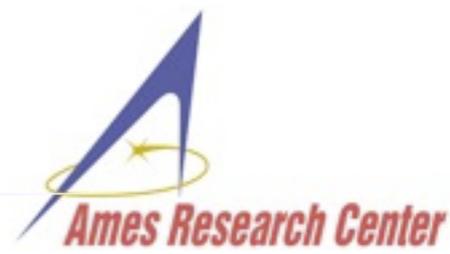
# SGOIDs - Same-ness Approaches



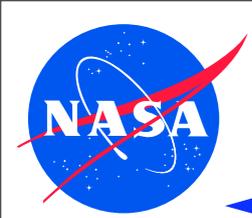
- ◆ approach 1: isomorphic heap graphs (Musuvathi,Dill 2004)
  - assumes canonical thread order
  - assumes everything that is order relevant is on the heap (StackFrames, Statics?)
  - requires additional (system) heap objects (e.g. class object container for ClassLoaders - only required at native side)
  - computes reference value from (unique) shortest access chain to global root object (breath first traversal of heap)
  - cannot be done at point of NEW execution, requires computation at PUTFIELD (which modifies the access chain) → problem with mixed SUT/System allocations
  - has to ensure non-overlapping reference values for objects, (sparse) reference values cannot be used as direct index into ElementInfo container
  - expensive



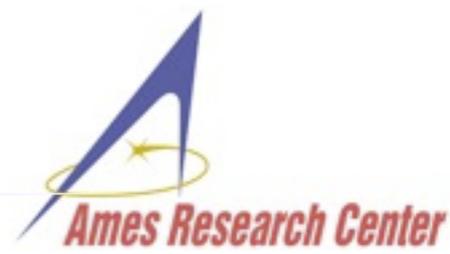
# SGOIDs - Same-ness Approaches



- ◆ approach 2: based on program state at point of allocation:
  - (1) allocated type (ClassInfo)
  - (2) allocating SUT thread state
    - ▶ allocating bytecode instruction
    - ▶ thread-local path (history)
  - (3) partial JPF state (Heap.newObject() caller)
  
- ◆ this is a model, there are variants (hashes vs. objects, stack depth etc.)
- ◆ single point of computation (native Heap.newObject(), Heap.newSystemObject() calls)
- ◆ has all the context (SUT and model checker)
  
- ◆ **use a configurable policy object to compute**



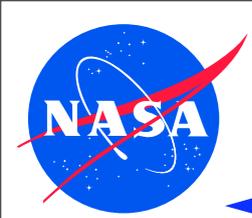
# SGOID Interface



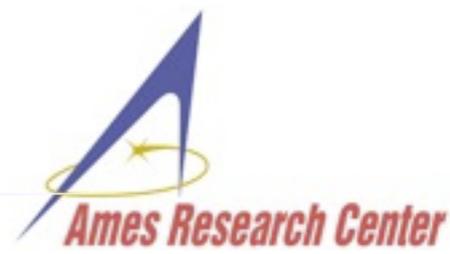
- ◆ hidden inside of already configurable Heap

```
interface Heap { ..  
    ElementInfo newObject (ClassInfo ci, ThreadInfo ti);  
    ElementInfo newSystemObject (ClassInfo ci, ThreadInfo ti, int anchor);  
  
    .. newArray(..), newSystemArray(..), newString(..), ..
```

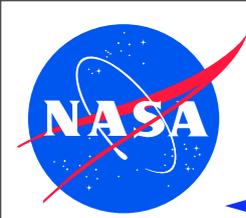
- ◆ from now on Heap implementations have to provide SGOIDs (Heap implementation harder)
- ◆ callers don't have to know how it works



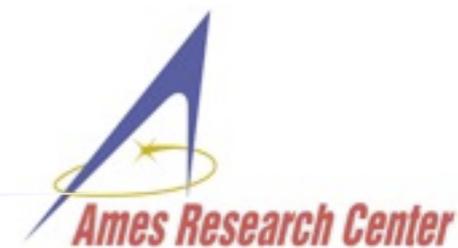
# SGOIDs - State Capture Example



- ◆ SUT allocation context:
  - type (ClassInfo)
  - allocating thread id
  - callstack abstraction of allocating thread (PC instructions)
  - Heap.newObject() caller (in case VM does related automatic allocs: Strings)
  
- ◆ System allocation context:
  - type
  - Heap.newObject() caller
  - anchor (tag that allows explicit classification)
  
- ◆ counter for same context (captures iteration state - can't be done with Stackframe slots since not clear which values reflect iteration)

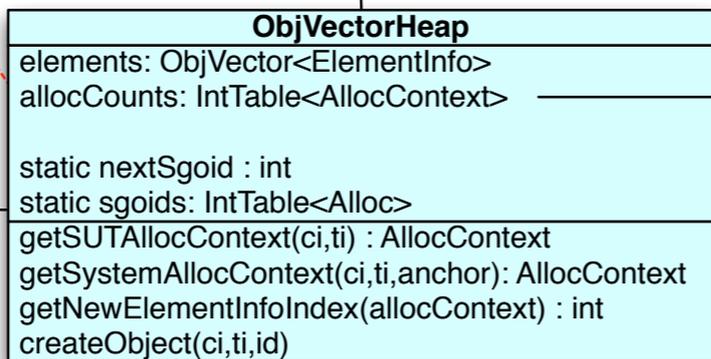
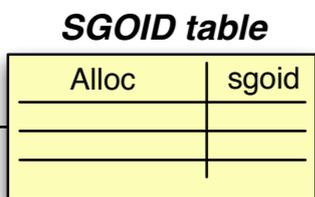
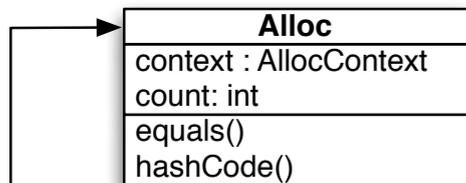
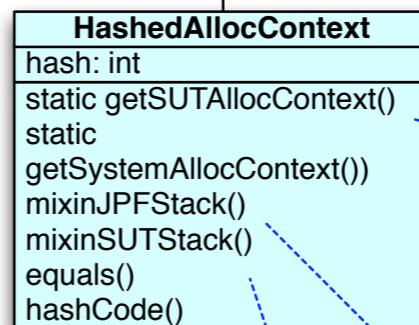
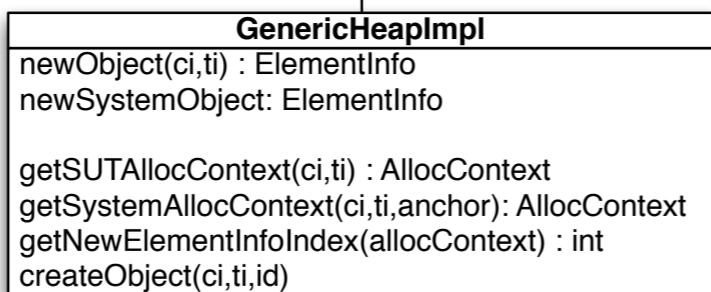
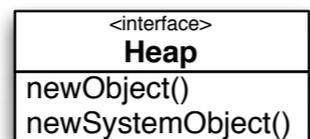


# SGOID Implementation Example

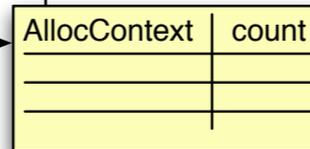


- ◆ SUT state: all current StackFrame PCs of allocating thread
- ◆ JPF state: StackTraceElement of newObject() caller
- ◆ both: iteration related state (stack/fields) captured with exec count

```
newObject (ClassInfo ci, ThreadInfo ti){
  AllocContext ctx =
    getSUTAllocContext( ci, ti);
  int idx = getNewElementInfoIndex( ctx);
  ElementInfo ei = createObject( .., idx);
}
```



AllocCount table



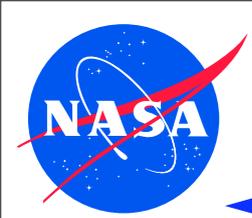
```
getSUTAllocContext(ClassInfo ci, ThreadInfo ti){
  h = hash(ci.getUniqueId());
  h = mixinSUTStack(h);
  h = mixinJPFStack(h,ti);
}
```

```
mixinJPFStack(){
  StackTraceElement[] ste =
    getStackTrace();
  StackTraceElement caller = ste[4];
  mixin(caller)
  ...
}
```

```
mixinSUTStack(int h, ThreadInfo ti){
  h = hash(ti.getId());
  for (StackFrame f : ti){
    Instruction pc = f.getPC();
    mixin(h, pc.getMethodInfo());
    mixin(h, pc.getInstructionIndex());
    mixin(h, pc.getBytecode());
  }
  ...
}
```

```
getNewElementInfoIndex(AllocContext ctx){
  int count = allocCounts.getInx(ctx).count;
  Alloc alloc = new Alloc(ctx,count);
  if (!sgoids.contains(alloc){
    int idx = ++nextSgoid;
    sgoids.put( alloc, idx);
    ...
  }
  assert elementInfos.get(idx) == null; ...
}
```

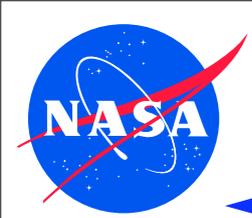
```
getSUTAllocContext(ClassInfo ci, ThreadInfo ti){
  return HashedAllocContext.getSutAllocContext(ci,ti);
}
```



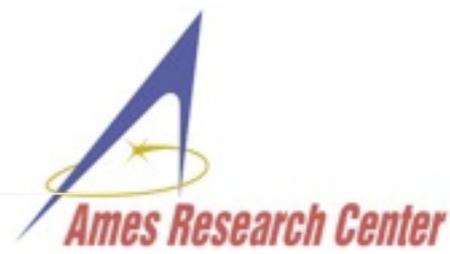
# SGOID - HashedAllocationContext



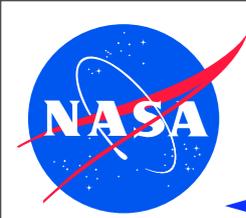
- ◆ assumptions
  - AllocCount table converges quickly (number and location of allocations?)
  - AllocContext elements are good hash components (low probability of collisions)
- ◆ drawbacks
  - needs to obtain host VM stacktrace for each allocation to automatically determine caller
  - depends on fixed call pattern for `Heap.newObject()` (can get called from other generic APIs such as `MJIEnv.newObject()`)
- ◆ stable (one failure does not cause failure in subsequent SGOID computation)



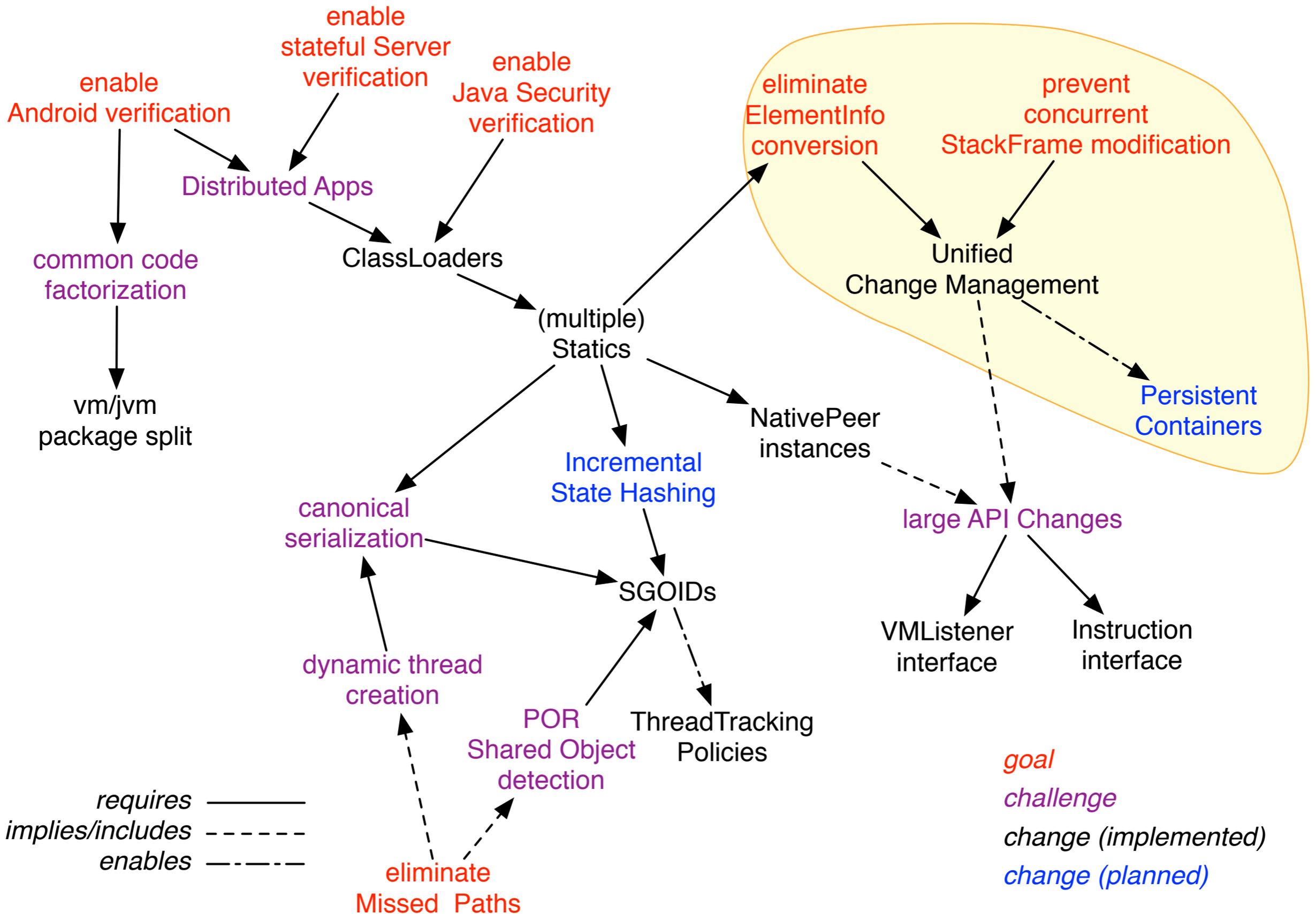
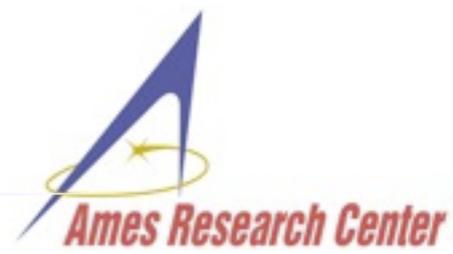
# SGOIDS - What if

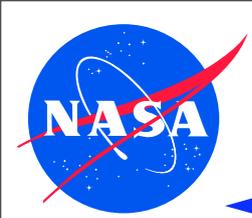


- ◆ erroneously same SGOID value for different objects
  - on same path with overlapping lifetime → caught in `Heap.newObject()` (ElementInfo index already taken) **critical consistency property**
  - different paths / non overlapping object lifetime:
    - ▶ additional states due to referencingThread set init
    - ▶ missed states due to premature matching
- ◆ erroneously different SGOID values for same object
  - ▶ additional states due to state leak (failed matching)
  - ▶ missed states due to referencingThread set init

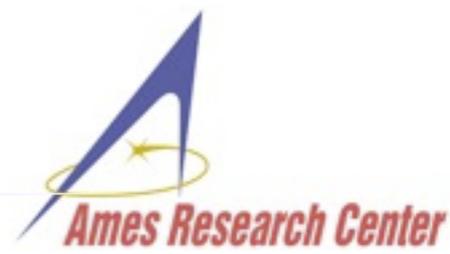


# v7 Change Map

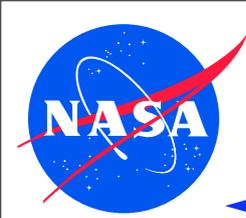




# Unified Change Management



- ◆ v6: state change management at different levels:
  - Fields, Monitor: ElementInfo (aggregate)
  - StackFrames: ThreadInfo (container)
  
- ◆ containers had to convert between Mementos and real objects when storing/restoring state
  
- ◆ post-store or concurrent modification by user code was possible
  
- ◆ v7: change management solely at container level
  - ElementInfo: Heap, Statics
  - StackFrame: ThreadInfo



# ElementInfo Change Management - v6



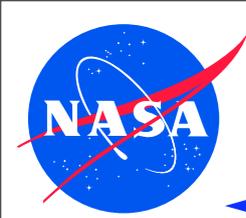
- ◆ class and object state management was hidden within ElementInfo

```
class ElementInfo {
    int attributes;
    Fields fields;
    ..
    void setIntField (FieldInfo fi, int newValue){
        if ((attributes & ATTR_FIELDS_CHANGED) == 0){
            fields = fields.clone();
            attributes |= ATTR_FIELDS_CHANGED;
        }
        fields.setIntValue(..)
        ..
    }
}
```

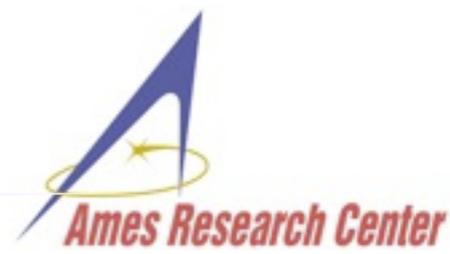
- ◆ clients were unaware of ElementInfo changes

```
class MyListener ..
    .. ei.setIntField( fi, 42);
```

- ◆ consequence: since ElementInfos are modified internally, can't directly store/restore Heap and StaticArea contents

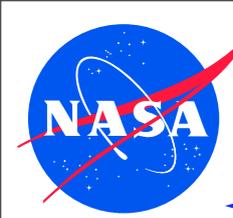


# ElementInfo Change Management - v6



- ◆ containers (Heap, StaticArea) had to convert ElementInfos to Memento<ElementInfo> instances during state storage, and vice versa during restore → persistent container types useless

```
class SparseClusterArrayHeap .. {  
    ..  
    static Transformer<ElementInfo, Memento<ElementInfo>> ei2mei = ..{  
        public Memento<ElementInfo> transform (ElementInfo ei){..  
            Memento<ElementInfo> mei = ei.getMemento();  
            ei.cachedMemento = mei;  
            return mei;  
            ...  
        }  
    static Transformer <Memento<ElementInfo>>, ElementInfo> mei2ei ..{..}  
  
    static class SCAMemento .. {  
        SCAMemento (SparseClusterArrayHeap heap){  
            snap = heap.getTransformedSnapshot( ei2mei);  
        }  
        Heap restore (Heap heap){ ..  
            heap.restoreTransformedSnapshot( snap, mei2ei);  
        }  
    }  
}
```



# ElementInfo Change Management - v7

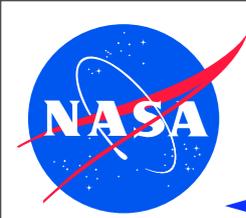


- ◆ change management lifted into container (Heap, Statics)
- ◆ ElementInfos get frozen upon state storage
- ◆ frozen ElementInfos can't be modified

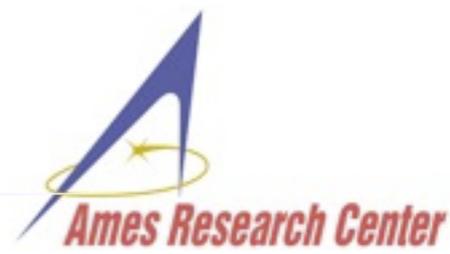
```
class ElementInfo .. {
    int attributes;
    void freeze() { attributes |= ATTR_IS_FROZEN; }

    void setIntField (FieldInfo fi, int newValue){
        if (isFrozen())
            throw new JPFException("attempt to modify frozen object");
        ...
    }
}

class ObjVectorHeap .. {
    static class OVMemento .. { ..
        OVMemento (ObjVectorHeap heap){
            for (ElementInfo ei : heap.elements)
                ei.freeze();
            snap = heap.elements.getSnapshot();
        }
    }
    Memento<Heap> getMemento() { return new OVMemento(this); }
}
```



# ElementInfo Change Management - v7

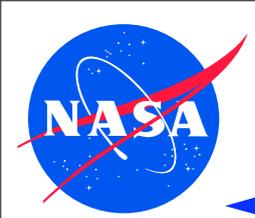


- ◆ clients have to explicitly ask container for modifiable /read-only objects
- ◆ implements/enforces copy-on-first-write

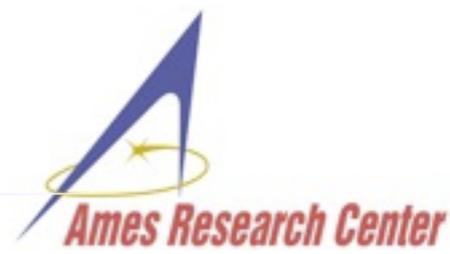
```
class ObjVectorHeap .. {
    ElementInfo get (int objref) { return elements.get(objref); }

    ElementInfo getModifiable (int objref){
        ElementInfo ei = elements.get(objref);
        if (ei.isFrozen()) {
            ei = ei.deepClone();
            elements.set(objref, ei);
        }
        return ei;
    }
}
```

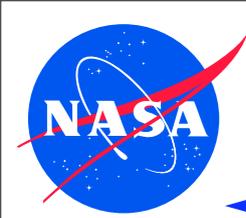
```
class MyListener .. {
    ..
    ElementInfo eiR = heap.get(objref);
    int d = eiR.getIntField("someField");
    ..
    ElementInfo eiRW = heap.getModifiable (int objref);
    eiRW.setIntField("someOtherField", 42);
}
```



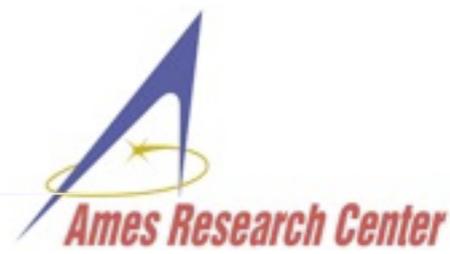
# ElementInfo Change Management - v7



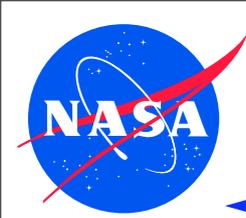
- ◆ Benefits
  - ElementInfos can be directly state stored (enables persistent Heap, Statics containers)
  - consistent with StackFrames: user has to explicitly declare intent, and hence is aware of changes
- ◆ primitive operation check overhead similar: instead of clone check, we have to check for modification of frozen ElementInfos



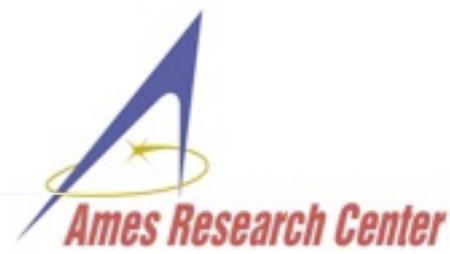
# ElementInfo Change Management - v7



- ◆ in theory complete ElementInfo deep cloning upon write request less memory efficient than (v6) separate Monitor / Fields cloning
- ◆ in reality not relevant:
  - fields change: very view objects have non-empty Monitors
  - monitor change: usually causes CG / complete state storage (single superfluous fields clone does not matter)  
caveat: large array store/restore problem not solved yet
- ◆ even without persistent containers, tests in medium state spaces (jpf-awt RobotManager, ~3000 states) showed slightly less memory consumption compared to v6 (no redundant Memento→ElementInfo restore required)
- ◆ persistent container types will save memory (and probably speed) - Heap/Statics container does not need separate snapshots anymore



# StackFrame Change Management - v6



- ◆ was hidden within ThreadInfo, but with exposure of potentially stored frames

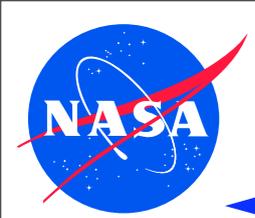
```
class ThreadInfo .. {
  StackFrame top;
  ...
  static class TiMemento .. {
    TiMemento (ThreadInfo ti) { .. savedTop = ti.top; ..}
    void restore (ThreadInfo ti) { .. ti.top = savedTop; ..}
    ...
  }
  void push (int v){
    if (top.hasChanged()) top = top.clone();
    top.push(v);
  }

  StackFrame getTopFrame() { return top; }
  ...
}

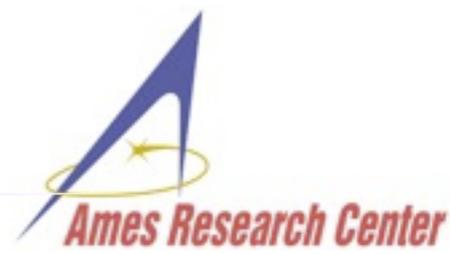
class JPF_x_y_MyPeer .. {
  ...
  StackFrame fr = ti.getTopFrame();
  .. ti.push(42); ..
  if (fr.peek() == 42) ..
}
```

CM relying on copy-on-first-write

different instances

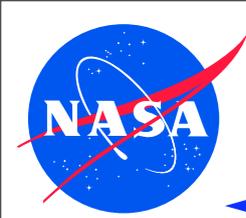


# StackFrame Change Management - v6

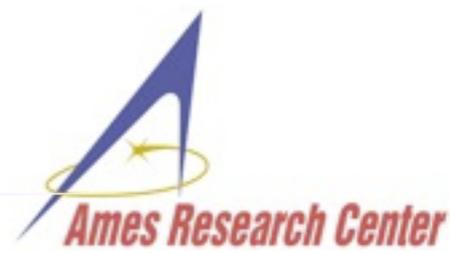


## ◆ Problems

- potential modification of stored StackFrames (rare since references mostly used within same transition / `Instruction.execute()`)
- erroneous StackFrame identity assumption (happened more often)
- API duplication (StackFrame API partially lifted into delegating `ThreadInfo`)
- Java bytecode dependencies (StackFrame) within general VM construct (`ThreadInfo`)



# StackFrame Change Management - v7



- ◆ change only through StackFrame (no push(), dup() etc. in ThreadInfo)
- ◆ modification checks if already state stored
- ◆ usage specific getters in ThreadInfo (StackFrame container)

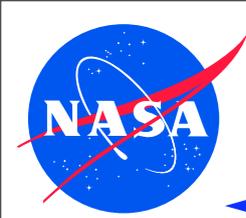
```
class StackFrame .. {
    int attributes;
    void freeze() { attributes |= ATTR_IS_FROZEN; }

    void push (int v) { // access ops only here
        if ((attributes & ATTR_IS_FROZEN) != 0) throw ...
        ..
    }
}

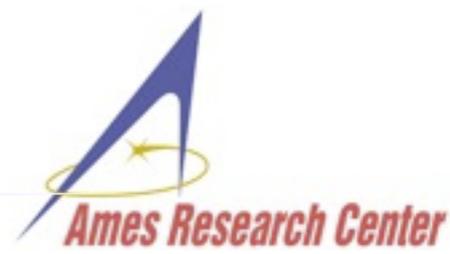
class ThreadInfo .. {
    static class TiMemento ..{
        TiMemento(ThreadInfo ti) {..for (StackFrame f : ti) f.freeze(); ..}
    }

    StackFrame getTopFrame() { return top; }

    StackFrame getModifiableTopFrame() {
        if (top.isFrozen()) top = top.clone();
        return top;
    }
}
// no more access ops delegation
```



# StackFrame Change Management - v7

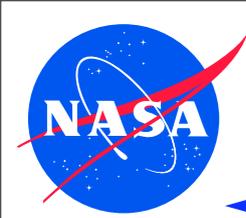


- ◆ clients need to explicitly state intention at point of StackFrame acquisition from container (ThreadInfo):

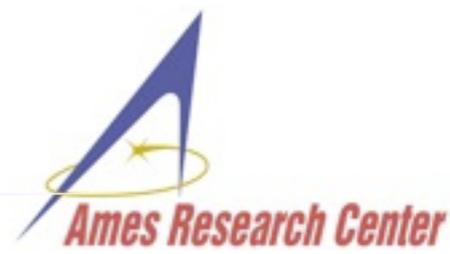
```
class MyListener .. {  
    ..  
    StackFrame frameR = ti.getTopFrame();  
    if (frameR.peek() == 42) ..  
    ..  
  
class MyPeer .. {  
    ..  
    StackFrame frameRW = ti.getModifiableTopFrame();  
    frameRW.push(42);  
    ..
```

- ◆ danger of identity confusion still there but reduced due to visibility

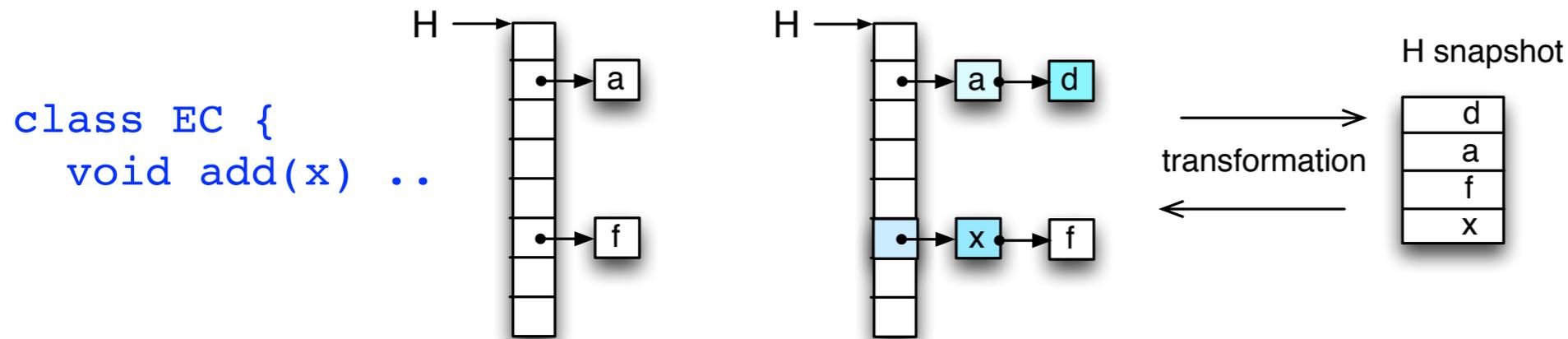
```
class MyPeer .. {  
    ...  
    StackFrame frame1 = ti.getModifiableTopFrame();  
    .. frame1.push(42) ..  
    StackFrame frame2 = ti.getTopFrame();  
    .. if (frame2.peek() == 42) // really?
```



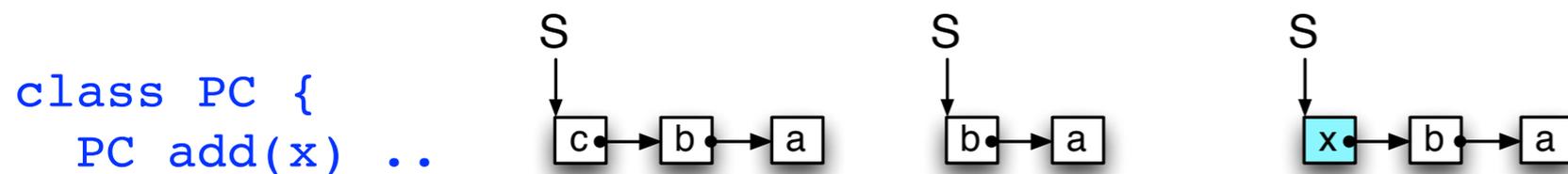
# State Storage - Persistent Containers



- ◆ ephemeral container: does not preserve previous state on modification
  - downside: requires deep copy or transformation when taking snapshots
  - upside: minimal modification (e.g. only changed value in array)

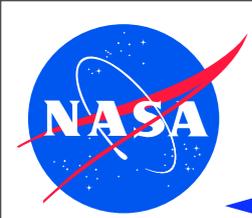


- ◆ persistent container: always preserves previous state
  - upside: snapshot is simply storing reference
  - downside: modification usually more expensive

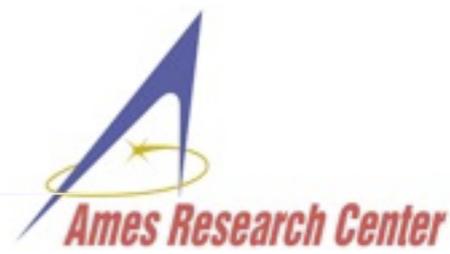


- ◆ can get a lot more sophisticated than Stacks:
  - .util.PersistentStagingMsbIntMap** (Heap, Statics)





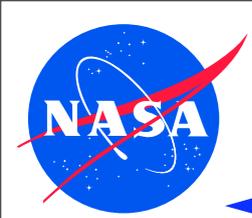
# VMListener Interface



- ◆ v6: single generic argument, notification specific data cached in JVM

```
class JVM {
    ThreadInfo lastThreadInfo;
    Instruction lastInstruction;
    ChoiceGenerator<?> lastCg;
    ...
    ThreadInfo getLastThreadInfo() { return lastThreadInfo; }
    ...
    void notifyExecuteInstruction (ThreadInfo ti, Instruction insn) {
        lastThreadInfo = ti; lastInstruction = insn;
        .. listener.executeInstruction(this);
        ...
    }
    void notifyChoiceGeneratorRegistered (ChoiceGenerator<?> nextCg) {
        lastCg = nextCg; // how intuitive
        ..
    }
}

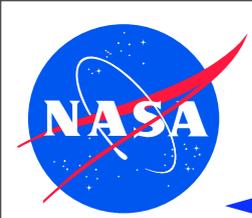
class MyListener .. {
    void executeInstruction (JVM vm) {
        ThreadInfo ti = vm.getLastThreadInfo();
        Instruction insn = vm.getLastInstruction(); // more like 'nextInsn'
        ...
    }
}
```



# VMListener Interface



- ◆ lastX fields in JVM considered harmful:
  - redundant (e.g. `lastThreadInfo == ThreadInfo.currentThreadInfo`)
  - unclear validity scope (some reset, others left after notification)
  - danger of using getters outside of notification methods
  - ambiguous naming (`lastChoiceGenerator`), changing semantics in each notification
- ◆ inefficient and noisy (who wasn't calling `getLastInstruction()` or `getLastThreadInfo()`)

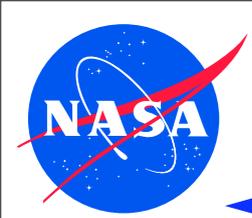


# VMListener Interface - v7



- ◆ v7: target data provided as explicit arguments
- ◆ different, notification type specific signatures
- ◆ VM argument still provided as fallback
- ◆ JVM.lastX fields and getters are gone

```
class MyListener .. {  
    ...  
    void executeInstruction (JVM vm,  
                            ThreadInfo currentTi,  
                            Instruction insnToExecute){  
        ...  
    void choiceGeneratorRegistered (JVM vm, ChoiceGenerator<?> nextCg){  
        ...  
    }
```



# Instruction Interface

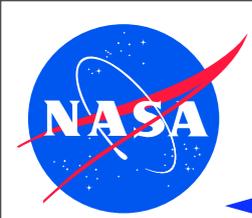


- ◆ ..since we are already changing Peer and Listener interfaces
- ◆ v6: fat interface for legacy reasons:

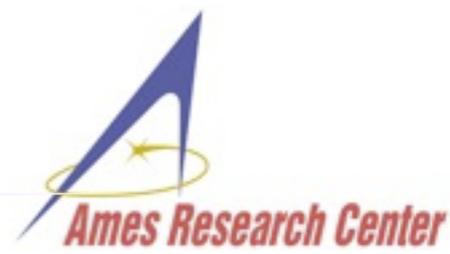
```
class Instruction {  
    ...  
    abstract Instruction execute(SystemState ss,  
                                KernelState ks,  
                                ThreadInfo ti);  
}
```

- ◆ KernelState not used, should be hidden (mostly for state management)
- ◆ SystemState rarely used, only for ChoiceGenerator APIs (can also be accessed through ThreadInfo, VM facades)
- ◆ v7: only ThreadInfo is passed in:

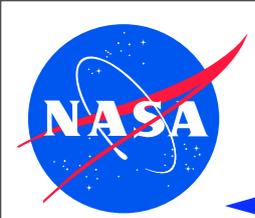
```
class Instruction {  
    ...  
    abstract Instruction execute(ThreadInfo ti);  
}
```



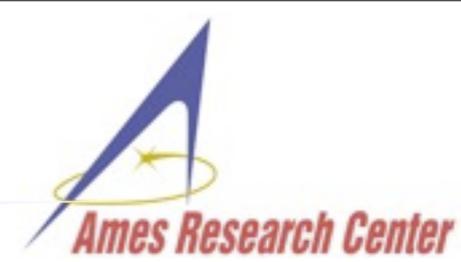
# VM / JVM separation

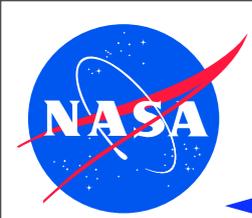


- ◆ Android VM (Dalvik) uses same Java VMSpec object, class and thread models
- ◆ has different bytecodes (abstract register machine) and binary class file format (\*.dex)
- ◆ need to factorize JPF into bytecode dependent / independent classes so that we don't have two systems with huge amount of overlap
- ◆ renames
  - `gov.nasa.jpf.jvm` → `gov.nasa.jpf.vm` (common classes)
  - `gov.nasa.jpf.classfile` → `gov.nasa.jpf.jvm.classfile`
  - `gov.nasa.jpf.jvm.bytecode` stays
  - `gov.nasa.jpf.jvm.JVM` → `gov.nasa.jpf.vm.VM`
  - ... and more to come (StackFrame needs to be separated)

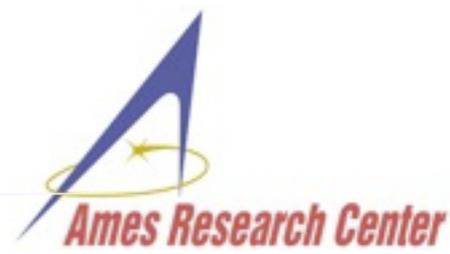


# Conclusions





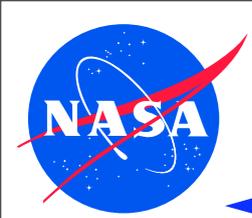
# Conclusions



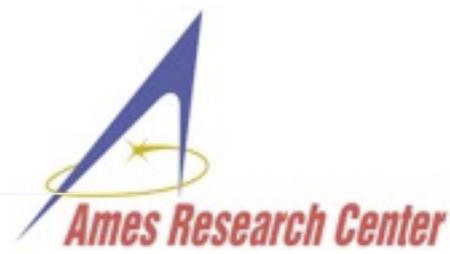
.. confused?

- ◆ slides are on <http://babelfish.arc.nasa.gov/trac/jpf/wiki/presentations/start> (jpf-v7-intro.\*)
- ◆ slides will be further elaborated over the next 2 months
- ◆ we will post a summary to the mailing list prior to v7 release
- ◆ hopefully, we can also throw in some scripts to port extensions (but help is welcome)

Thank You!



# Conclusions



.. confused? I'm with you

- ◆ slides are on <http://babelfish.arc.nasa.gov/trac/jpf/wiki/presentations/start> (jpf-v7-intro.\*)
- ◆ slides will be further elaborated over the next 2 months
- ◆ we will post a summary to the mailing list prior to v7 release
- ◆ hopefully, we can also throw in some scripts to port extensions (but help is welcome)

Thank You!